Trabajo Práctico 1

Sistemas Operativos
Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Chibana, Christian	586/13	christian.chiba93@gmail.com
De Carli, Nicolás	164/13	nikodecarli@gmail.com
Minces Müller, Javier	231/13	javijavi1994@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. Introducción

Un sistema operativo multitarea permite ejecutar varios procesos compartiendo recursos, específicamente el procesador. Para esto, es necesario un componente que decida, por cada ciclo de clock, qué proceso debe ejecutarse.

Este componente es el scheduler, que además debe encargarse de poner a ejecutar el proceso y, si es necesario, desalojar el proceso anterior.

Los procesos son instancias de un programa, es decir que necesitan toda la información sobre su contexto: memoria, registros, código fuente. Por eso, cambiar el proceso que se está ejecutando en un $core$ tiene un costo temporal. También tiene un costo adicional migrar un proceso de un $core$ a otro del procesador.

Los procesos pueden hacer llamadas bloqueantes, generalmente para ejecutar operaciones de entrada/salida.

En este trabajo utilizaremos el simulador $simusched$, dado por la cátedra, para simular un scheduler. En este simulador, se 'avisa' al scheduler cada vez que llega una tarea nueva (load) o una tarea manda un $unblock$. Cuando termina un ciclo de clock, se llama al scheduler una vez para cada $core$, indicando si la tarea que se estaba ejecutando terminó, ejecutó una llamada bloqueante, o no. El scheduler debe indicar que tarea se sigue ejecutando.

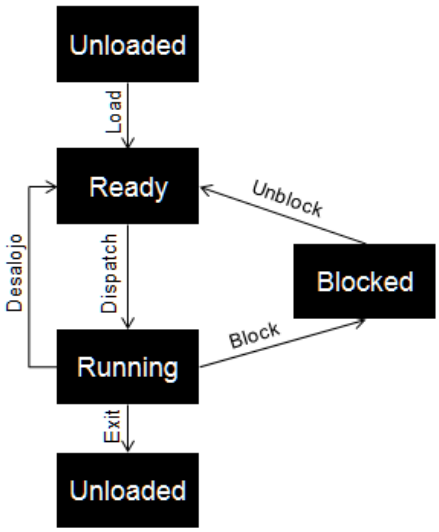

Figura 1: Máquina de estados para este simulador

Los algoritmos ya programados son FCFS (First Come, First Serve), en el que los procesos se ejecutan hasta terminar por orden de llegada y SJF (Shortest Job first), en el que la tarea de menor duración se ejecuta hasta el final y luego se ejecuta el siguiente con menor duración.

2. Desarrollo

2.1. Tipos de Tareas

2.1.1. Ejercicio 1: TaskConsola

En este ejercicio se pedía simular una tarea interactiva con n llamadas bloqueantes y que cada llamada tome una duración al azar entre dos valores bmin y bmax. Esto lo implementamos a partir del uso de la función uso_IO y la función $rand()$ para generar lo pedido.

Este es el comportamiento de una de estas tareas, simulado con el algoritmo FCFS (First Come, First Serve). Se pueden ver cinco llamadas bloqueantes en el diagrama de Gantt, con una duración de entre 2 y 3 ciclos.

Figura 2: Un TaskConsola

2.1.2. Ejercicio 2

En este ejercicio se pedía simular un lote de tres tareas, una intensiva en CPU y dos de tipo TaskConsola, con uno, dos y tres $cores$. Obtuvimos estos resultados:

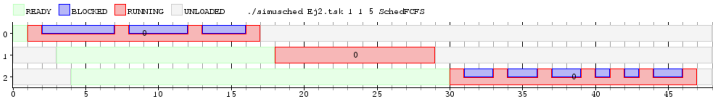

Figura 3: Un TaskCpu y dos TaskConsola con 1 núcleo

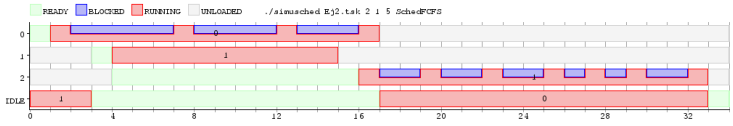

Figura 4: Un TaskCpu y dos TaskConsola con 2 núcleo

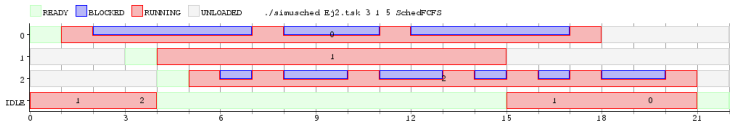

Figura 5: Un TaskCpu y dos TaskConsola con 3 núcleos

En los gráficos se puede ver como funciona el algoritmo FCFS, en donde se observa que dependiendo de la llegada de las tareas, estas se ejecutan en ese orden de llegadas sin realizar desalojos por bloqueos.

La primera tarea llamada es la 0, que ejecuta LOAD en el ciclo 0, luego se ejecuta la 1, que lo hace en el ciclo 3, y por último la 2.

Se puede ver que, como este método no tiene desalojo, las tareas siguen consumiendo tiempo de CPU cuando están bloqueadas. Además, como no se tiene en cuenta la duración de las tareas, puede pasar que una tarea corta tenga que esperar a que se termine de ejecutar una más larga, como ocurre en el ejemplo de 1 $core$.

Sin embargo, se minimiza el tiempo de migración y de cambio de contexto, ya que las tareas se ejecutan en un mismo $core$ hasta que terminan. Sin embargo, como se puede ver en la Figura 4, un $core$ puede pasar mucho tiempo idle mientras espera a que llegue una nueva tarea.

Por último, al comparar los tres gráficos, se puede ver que al aumentar el número de $cores$ se reduce notablemente la ejecución total de las tres tareas ya que se pueden ejecutar en simultáneo cada una en un $core$ distinto.

2.1.3. Ejercicio 6: TaskBatch

Este tipo de tareas ejecuta n llamadas bloqueantes en momentos aleatorios, cada una de un ciclo de duración, y en total utiliza m ciclos de CPU. Para implementar esto se utiliza un vector v de números pseudoaleatorios diferentes ordenados con una distribución $U \sim (0, m)$, y se lo recorre usando la CPU $(v_i - v_{i-1} - 1)$ ciclos y bloqueándose un ciclo alternativamente

Este es el comportamiento de una TaskBatch con $m = 25$ y $n = 6$ en el simulador FCFS:

Figura 6: TaskBatch

2.2. Scheduler Round Robin

Un scheduler 'Round Robin' tiene un quantum asignado para cada *core*. Este valor indica por cuantos ciclos de clock una tarea puede utilizar el CPU antes de ser desalojada. Si la tarea se bloquea, también es desalojada.

2.2.1. Ejercicio 3

Para implementar esto utilizamos la cola de la librería *std* para las tareas, junto con dos vectores, uno para el quantum de cada *core* y otro para el tiempo transcurrido de cada quantum.

En cada ciclo, el scheduler observa que *cores* llegaron al final del quantum, desalojando la tarea actual si eso ocurre. Lo mismo hace si la tarea se bloquea. Al desalojar, la agrega al final de la cola y la reemplaza por la primera tarea en la cola. Además, pone en 0 el valor de quantum del core correspondiente.

2.2.2. Ejercicio 4

Obtuvimos los siguientes resultados con un bloque de tareas en el scheduler 'Round Robin'.

```
TaskCPU 12
@1:
TaskIO 5 2
@3:
TaskAlterno 1 10
```

Figura 7: SchedRR con 1 Core

En esta Figura 7 se tiene 1 *core* con quantum 1 y costo de cambio de contexto igual a 1 ciclo. Se puede observar que se ejecuta la tarea que más tiempo de espera tuvo, ya que en la cola se pushea la tarea cuando pasa a estar ready. Se puede observar que las tareas tienen un quantum de 1 ciclo, y que los bloqueos tienen desalojo por lo tanto no se les otorga ciclos de CPU. También se puede observar que si hay una sola tarea a ejecutar, se le otorga todo el tiempo de procesamiento hasta que termine o hasta que una nueva tarea pase a estar ready. Esto es lo que se espera de un algoritmo *Round-Robin* con un solo *core*.

```
TaskCPU 10
TaskCPU 10
TaskCPU 10
```

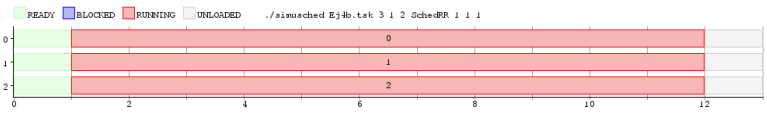

Figura 8: SchedRR con 3 Core y 3 tareas

En esta Figura 8 se puede observar que al ser la misma cantidad de tareas que de *cores*, las tareas se ejecutan en simultáneo una en cada *core* sin desalojos. Pero al insertar otra nueva tarea ocurre esto:

```
TaskCPU 10
TaskCPU 10
TaskCPU 10
TaskCPU 10
```

Figura 9: SchedRR con 3 Core y 4 tareas

Al insertar esta nueva tarea se genera la necesidad de hacer desalojos para atender esta nueva tarea. Se siguen ejecutando en simultáneo las tareas, pero respetando el quantum de cada $core$. Por último, se observa que en cada tarea, siempre se genera un cambio de $core$ en el cual ejecutar, por lo tanto, esto hace que se sume al costo de cambio de contexto, el costo de migración. Esto último se genera debido a que cuando un $core$ termina de ejecutar una tarea durante su quantum, encuentra otra que en la cola se encuentra antes que la tarea recién ejecutada. En este caso, un algoritmo de 'Round Robin' sin migraciones entre $cores$ mejoraría la ejecución total de las tareas.

2.2.3. Ejercicio 7: Quantum óptimo

Para encontrar el valor óptimo de quantum para 1 costo de cambio de contexto y 2 de costo de migración, experimentamos con diez ejecuciones del siguiente lote de tareas:

```
TaskBatch 20 3
TaskBatch 20 6
TaskBatch 20 8
TaskBatch 20 7
TaskBatch 20 2
```

Utilizamos como métricas el $throughput$, definido como la cantidad de procesos que finalizan su ejecución por unidad de tiempo, y el $waitingtime$, que es la cantidad de ciclos que una tarea pasa en estado $ready$ en promedio.

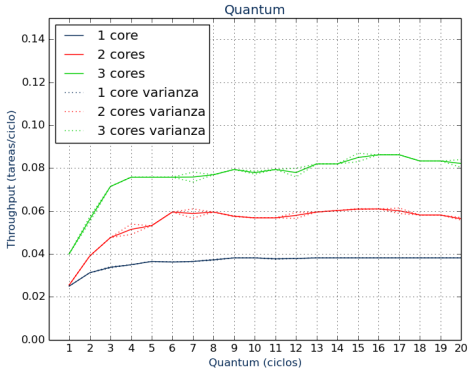

Figura 10: $Throughput$

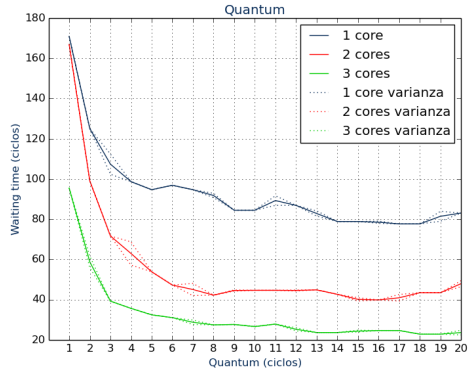

Figura 11: Waiting time promedio

Lo que se puede ver en los gráficos es que el resultado tiende a ser mejor para ambas métricas cuanto mayor sea el quantum, hasta estabilizarse, lo que ocurre alrededor de un valor de 13 ciclos. A partir de

este momento, no hay diferencia en el resultado porque las tareas elegidas no tienen intervalos de más de 13 ciclos sin bloqueos.

2.2.4. Ejercicio 8

Una variante de este método consiste en fijar cada proceso a un $core$, de forma que siempre se ejecute en este. En vez de una cola global, se tiene una para cada $core$. Con esto se busca eliminar el costo de migración de procesos.

Para comparar este método con el anterior, elegimos un lote de cinco tareas TaskBatch y cinco tareas TaskCPU. Luego elegimos tres de los cuatro parámetros del simulador y los fijamos de la siguiente forma:

Número de cores	2
Costo de cambio de contexto	1
Costo de migración	2
Duración del quantum	2

Mientras que el parámetro restante lo hicimos variar entre 1 y 4. Luego de diez ejecuciones, obtuvimos estos resultados de $throughput$:

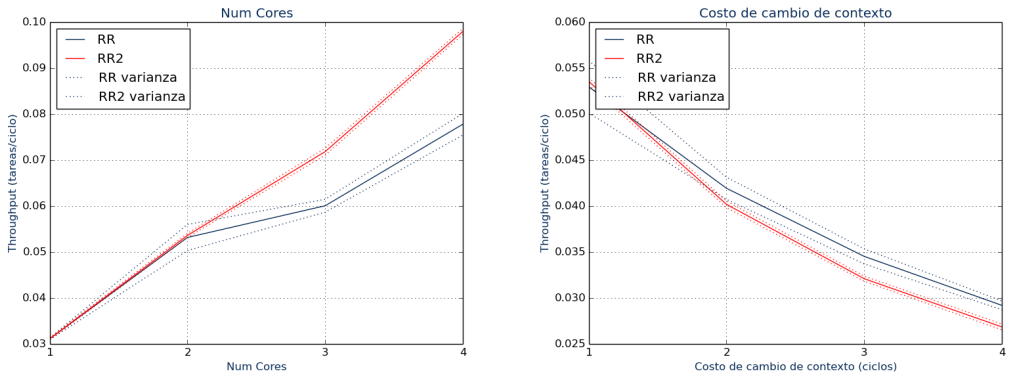

Figura 12: Resultado según número de cores y costo de cambio de contexto

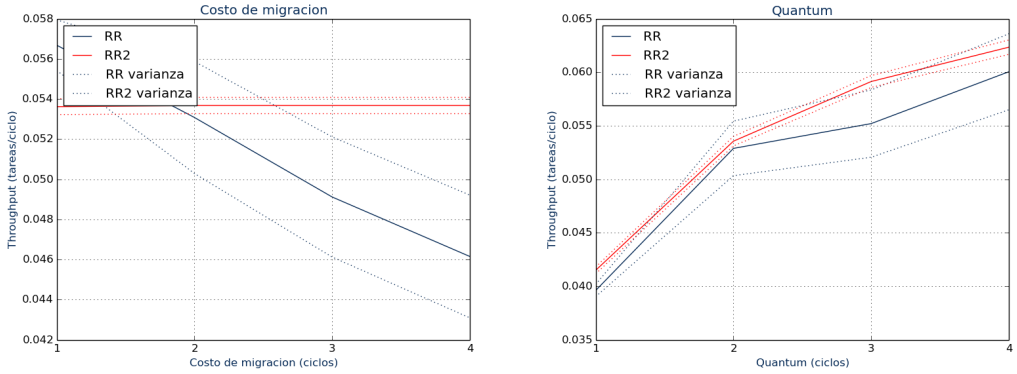

Figura 13: Resultado según costo de migración y valor del quantum

En las figuras 12 y 13 se puede observar que en cuanto al aumento de $cores$ y al aumento de quantum, el scheduler RR2 tiene un mayor throughput, para los valores elegidos. Esto ocurre ya que se evita el costo de cambio de contexto, y al aumentar los $cores$ el scheduler RR dispone de más $cores$ por lo que puede realizar más cambios de contexto.

En cuanto al aumento del costo de migración, se puede observar que cuando el costo es 1, el scheduler RR tiene un mayor $throughput$ que el scheduler de RR2, ya que evita situaciones en las que, por ejemplo, dos tareas se ejecutan en un core mientras el resto está idle. Para valores mayores, el algoritmo de RR2 no varía su $throughput$ ya que no realiza migraciones, mientras que el del RR baja.

Por último, cuando se aumenta el costo de cambios de contexto, el $throughput$ del RR2 se hace menor ya que al no poder realizar migraciones, genera más cambios de contexto en cada $core$ que el RR. Esto se debe a situaciones como tener un core ejecutando varias tareas mientras otros están idle.

Los resultados obtenidos para $waiting\ time$ fueron los siguientes:

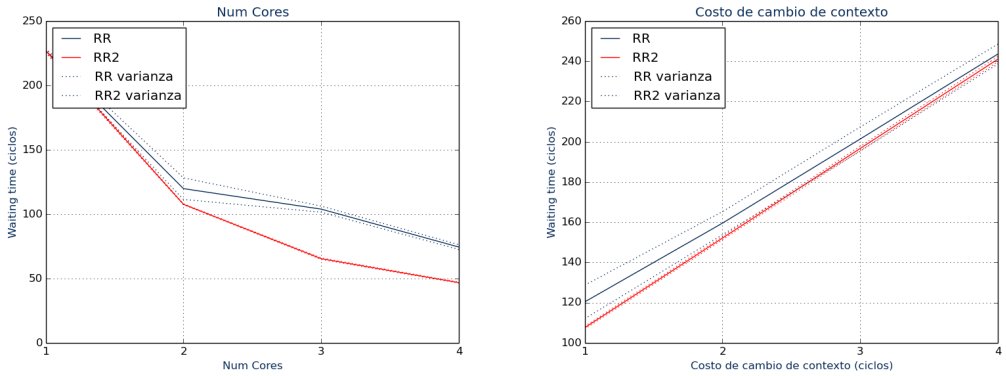

Figura 14: Resultado según número de cores y costo de cambio de contexto

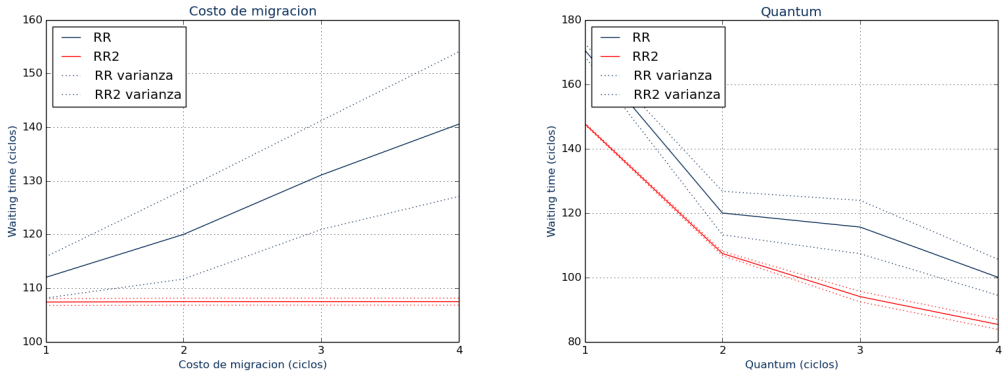

Figura 15: Resultado según costo de migración y valor del quantum

Se puede observar, en estos gráficos de $waiting\ time$, que en todos los casos el scheduler RR2 obtuvo un menor $waiting\ time$ que el scheduler RR. Esto muestra que el algoritmo sin realizar migraciones es mejor en cuanto a esta métrica. Vemos también que cuando se aumenta el costo de cambio de contexto, el $waiting\ time$ del scheduler RR2 crece de una forma más rápida que el scheduler de RR, esto se debe a que, como ya se dijo, al no poder realizar migraciones, el RR2 realiza más cambios de contexto.

2.3. Schedulers propuestos por Liu y Layland

2.3.1. Ejercicio 5

Los autores intentan crear un conjunto de reglas que permita elegir de manera eficiente en cada momento qué tarea debe ser ejecutada dentro de un sistema, es decir, un scheduler. Cada tarea tiene un tiempo fijo que tarda en ser ejecutada si es computada por el procesador de manera ininterrumpida, además se ejecutan periódicamente, o sea tienen una constante temporal c y una lista infinita de momentos $x_1, x_2, x_3 \ldots$ tal que $x_1 = c, x_2 = x_1 + c, x_3 = x_2 + c$, etc. Y en cada uno de ellos se requiere su ejecución nuevamente. Las tareas son independientes unas de otras, es decir ninguna puede activar el pedido de ejecución de otra. Los autores buscan diseñar un scheduler basado en prioridades, es decir, que dado un momento cada tarea tenga un valor numérico asignado y se ejecute la que posea el mínimo de todos. Luego es discutida la manera de asignación de dichos valores, una opción es que cada tarea tenga una prioridad fija, la otra es que en cada momento la tarea para la cual falte menos tiempo en ser

pedida nuevamente tenga la prioridad más alta, en contraste, la de menos prioridad será la que le falte más tiempo para que su ejecución sea pedida de nuevo.

Los autores plantean que un scheduler con prioridades fijas, es decir que no varíen con el tiempo, utiliza como máximo un $70\,\%$ del procesador, y por lo tanto introducen una forma dinámica de asignar prioridades en cada momento con el fin de aprovechar el poder de cómputo del sistema lo máximo posible.

La idea del teorema 7 es que un conjunto de tareas es realizable en un scheduler con prioridades dinámicas si y solo si la demanda total de tareas no excede el tiempo disponible de procesador. De esto podemos desprender que un conjunto de tareas puede ser ejecutado de forma óptima siguiendo la idea de prioridades dinámicas si puede ser ejecutado en cualquier otro scheduler. Esto en parte trata de asegurar de que si se corren tareas en algún sistema de forma satisfactoria, entonces no va a ocurrir en un scheduler de prioridades dinámicas que una tarea no haya sido ejecutada por completo en el momento en que es pedida de nuevo.

Para implementar el scheduler fijo, utilizamos una cola de prioridad, donde la prioridad está dada por el período de cada tarea. Para implementar el dinámico, recurrimos a una lista ordenada, que permite acceder a cualquier elemento. La intención es guardar en la lista el tiempo restante para el deadline de cada tarea, acutalizándolo cada tick. En caso de overflow, considera que el tiempo restante es negativo.

2.3.2. Ejercicio 9

Un ejemplo de un lote de tareas que es factible para el scheduler dinámico pero no para el fijo es el siguiente:

```
&05,4,1
&15,6,2
```

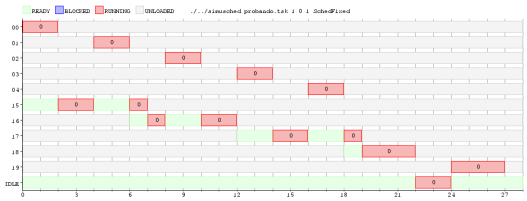

Figura 16: Scheduler fijo

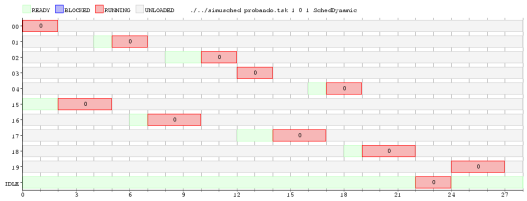

Figura 17: Scheduler dinámico

Se puede ver que se produce overflow en el ciclo 6 del scheduler fijo, pero no se produce en el dinámico. Esto es porque el fixed desalojó a la tarea que se estaba ejecutando cuando llega una de menor período en el ciclo 4, aunque su deadline estuviera más cercano

3. Conclusiones

El scheduler que conviene utilizar siempre depende de una gran cantidad de factores. Por ejemplo, se vio que, para las métricas que usamos, el scheduler Round Robin óptimo es análogo a un schedFCFS que desaloja las tareas cuando se bloquean. Sin embargo, el round robin muestra ventajas que no medimos en el desarrollo. Por ejemplo, evita dejar tareas sin empezar a ejecutar hasta que alguna termine. Por

otra parte, en nuestra comparación entre el Round Robin con y sin migración vimos que el algoritmo sin migración era mejor en los casos en donde se aumentaban el número de $cores$, los costos de migraciones, y los quantums, pero no en el caso en donde se aumentaba el costo de cambio de contexto. En general, el algoritmo sin migración es mejor, pero solo suponiendo un costo de migración alto.

Los algoritmos propuestos en el paper son óptimos, pero bajo la suposición de que las tareas son periódicas, lo que no se da siempre. Lo que se puede concluir de esto es que para elegir un algoritmo, hay que tener en cuenta no solo todos los parámetros del simulador, sino también las características de las tareas a ejecutar.

4. Referencias

- Apuntes de clase
- *Liu, Chung Laung, and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (JACM) 20.1 (1973): 46-61*