Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

TP1 - Scheduling

Sistemas Operativos

Primer Cuatrimestre de 2015

Apellido y Nombre	LU	E-mail
Cisneros Rodrigo	920/10	rodricis@hotmail.com
Rodríguez, Agustín	120/10	agustinrodriguez90@hotmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Contents

1 Introducción

En este Trabajo Práctico estudiaremos diversas implementaciones de algoritmos de scheduling. Haciendo uso de un simulador provisto por la cátedra podremos reprensentar el comportamiento de estos algoritmos. Implementaremos dos Round-Robin, uno que permite migración de tareas entre núcleos y otro que no y a través de experimentación intentaremos comparar ambos algoritmos. Asimismo, basándonos en un paper implementaremos una versión del algoritmo *Lotery* y mediante experimentos intentaremos comprobar ciertas propiedades que cumple el algoritmo.

2 Desarrollo y Resultados

3 Parte I – Entendiendo el simulador simusched

3.1 Ejercicios

- **Ejercicio 1** Programar un tipo de tarea TaskConsola, que simulara una tarea interactiva. La tarea debe realizar n llamadas bloqueantes, cada una de una duracion al azar 1 entre bmin y bmax (inclusive). La tarea debe recibir tres parametros: n, bmin y bmax (en ese orden) que seran interpretados como los tres elementos del vector de enteros que recibe la funcion.

- **Ejercicio 2** Escribir un lote de 3 tareas distintas: una intensiva en CPU y las otras dos de tipo interactivo (TaskConsola). Ejecutar y graficar la simulacion usando el algoritmo FCFS para 1, 2 y 3 nucleos.

3.2 Resultados y Conclusiones

3.2.1 Ejercicio 1

Dada la simpleza del código, optamos por mostrar nuestra implementación, en vez de comentarlo detalladamente.

Realizamos un ciclo de i ¡ params[0], donde utilizamos la función dada por la catedra, uso_IO a la cual le pasamos el pid correspondiente y un entero ciclos que es el valor random obtenido entre *bmin* y *bmax*

```
ciclos = rand() % (params[2] - params[1] + 1) + params[1];
```

A continuacion, el codigo mencionado:

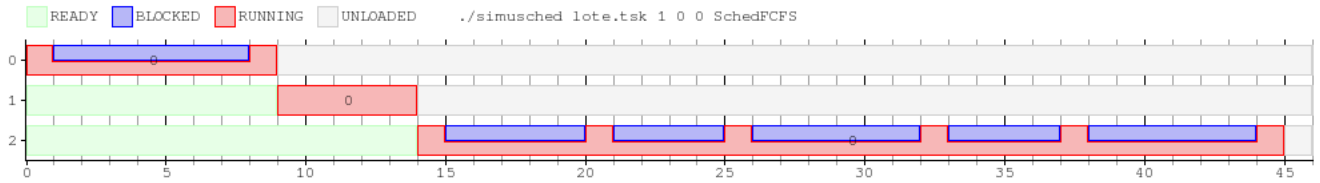
```
void TaskConsola(int pid, vector<int> params) {
     int i, ciclos;
     for (i = 0; i < params[0]; i++) {
            ciclos = rand() % (params[2] - params[1] + 1) + params[1];
            uso_IO(pid, ciclos);
     }
}
```

3.2.2 Ejercicio 2

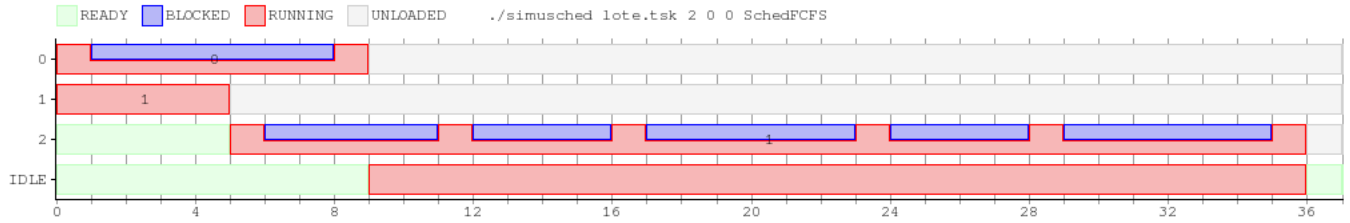
Para este punto, utilizamos el siguiente lote de tareas:

```
TaskConsola 1 4 8
TaskCPU 4
TaskConsola 5 3 6
```

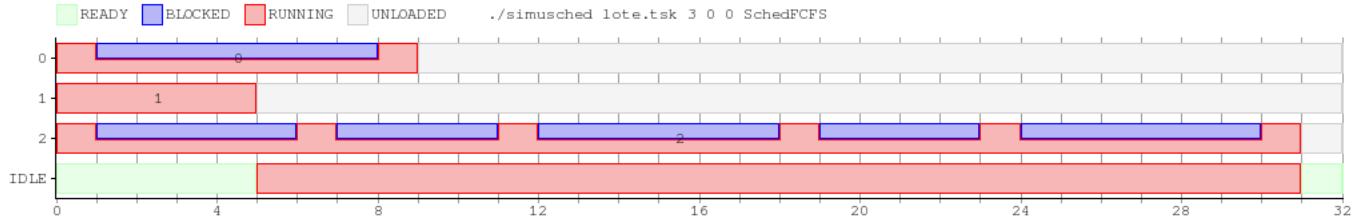
El mismo, presenta una tarea de uso intensivo $TaskCPU$ que dura unos 4 ticks, y otras dos interactivas, las cuales se bloquean 1 y 5 ticks respectivamente con una duración de entre 4 y 8 para la primera y 3 y 6 para la segunda.
A continuacion, los respectivos graficos de mediciones



*Lote*1 Scheduler FCFS - 1 core



*Lote*1 Scheduler FCFS - 2 core



*Lote*1 Scheduler FCFS - 3 core

Debido a las tareas de tipo TaskConsola, se observa que en los tres casos la duración de cada bloqueo es distinta.

A modo de análisis, se puede observar por medio de los gráficos como aumenta el paralelismo a mayor cantidad de núcleos. En este scheduler en particular esto ayuda de gran manera al rendimiento del sistema, puesto que un nucleo podrá ejecutar otra tarea recién cuando haya terminado la anterior. Las consecuencias de este comportamiento son visibles en los 3 graficos. Agregando un core más, el tiempo que se tarda en ejecutar por completo todas las tareas de reduce casi a la mitad.

4 Parte II: Extendiendo el simulador con nuevos schedulers

4.1 Ejercicios

- **Ejercicio 3** Completar la implementacion del scheduler Round-Robin implementando los metodos de la clase SchedRR en los archivos sched rr.cpp y sched rr.h. La implementacion recibe como primer parametro la cantidad de nucleos y a continuacion los valores de sus respectivos quantums. Debe utilizar una unica cola global, permitiendo ası la migracion de procesos entre nucleos.

- **Ejercicio 4** Diseñar uno o mas lotes de tareas para ejecutar con el algoritmo del ejercicio anterior. Graficar las simulaciones y comentarlas, justificando brevemente por que el comportamiento observado es efectivamente el esperable de un algoritmo Round-Robin.

- **Ejercicio 5** A partir del artıculo Waldspurger, C.A. and Weihl, W.E., Lottery scheduling: Flexible proportional-share re- source management. Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation – 1994. diseñar e implementar un scheduler basado en el esquema de loterıa. El constructor de la clase SchedLottery debe recibir dos parametros: el quantum y la semilla de la secuencia pseudoaleatoria (en ese orden). Interesa implementar al menos la idea basica del algoritmo y la optimizacion de tickets compensatorios (compensation tickets). Otras optimizaciones y refinamientos que propone el artıculo seran opcionales siempre que, en cada caso, se explique brevemente por que la optimizacion no se considero relevante a los efectos de este trabajo.

4.2 Resultados y Conclusiones

4.2.1 Ejercicio 3

Para completar la implementación del scheduler Round-Robin y que su comportamiento sea correcto, hicimos uso de diversas estructuras de datos cuya composición y uso describimos a continuación.

- Una cola global FIFO nombrada q, que contiene los pid de las tareas activas no bloqueadas y cuyo tope representa a la próxima tarea a correr. Utilizando una cola FIFO podemos obtener el comportamiento deseado, ya que al desalojarse una tarea por consumir su quantum esta misma será agregada nuevamente a la cola, quedando al final de ésta y generando el ciclo que buscamos. Al ser además la única cola para todos los cores, no se restringe a una tarea a ser ejecutada por un único núcleo, permitiendo así la migración entre nucleos.

- El vector *cores* contiene en su elemento i y el pid correspondiente a la tarea que está corriendo en el core $i + 1$. Inicializamos todos sus elementos en -1 (que se corresponde con la Idle Task) para reconocer que no se han cargado tareas en los núcleos de procesamiento.

- De la misma manera, el vector *quantum* contiene en la posición i el quantum definido para el núcleo y el vector *quantumActual*, contiene la cantidad de ticks que le quedan desde que se cargó la tarea en el core. En conjunto, ambas estructuras nos permiten determinar cuándo se consumió el quantum de una tarea, de manera tal que podamos desalojarla.

Ademas, tomamos ciertas decisiones en esta implementación las cuales detallamos a continuacion:

- Si una tarea se encuentra bloqueada cuando se produce el tick del reloj, esta misma es desalojada de la cola global, y agregada en un lista de *bloqueados*. A su vez, será reseteado el quantum, se le dara inicio a la proxima tarea que se encuentre ready y cuando el sistema operativo, nos envie una señal de unblock, la tarea desalojada regresara al final de la cola global.

4.2.2 Ejercicio 4

El algoritmo de scheduling Round-Robin se basa en asignar a las tareas un tiempo determinado de procesamiento en porciones equitativas con un orden cíclico, llamado *quantum*. Esto se hace definiendo para cada núcleo de procesamiento un valor de *quantum* y cada tarea que corre en ese núcleo lo hará a lo

sumo por un tiempo igual a éste. Si la tarea consume su quantum pero no terminó su ejecución, se la envia a la cola global y se asigna el procesador a la siguiente tarea, en caso de que hubiese, que correrá a lo sumo por la cantidad de *quantum* indicada. Una vez que todas las tareas corrieron, se vuelve a asignar tiempo de procesamiento a la primera tarea, de ahí el comportamiento ciclico circular del algoritmo.

A su vez, puede ocurrir que una tarea no consuma todo su *quantum*. Ya sea porque la tarea terminó su ejecución o bien se bloqueó haciendo uso de entrada/salida.

En caso de haber terminado, el algoritmo se pone a correr la tarea siguiente de acuerdo al orden circular que se establecio y la tarea que terminó se desalojara por completo y no sera nuevamente considerada.

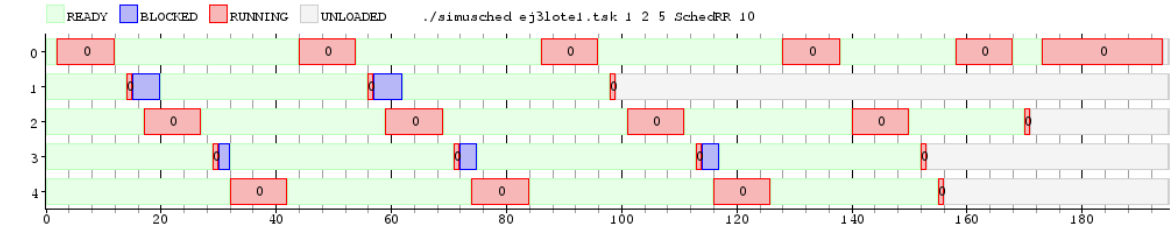
En caso que se haya bloqueado, esta mimsa dejará de ser considerada hasta que se desbloquee. Mientras tanto, seguiran corriendo las demas tareas que no esten bloqueadas respetando el orden ciclico. Cuando esta tarea se desbloquee puede no respetar el orden que se habia establecido. Por ej. en nuestra implementacion si la tarea esta bloqueada al recibir un tick, esta misma se la quita de la ejecucion y una ves desbloqueada pasara al final de la cola global.

Para poder verificar si el comportamiento era el deseado de nuestra implementación del algoritmo en cuestión, desarrollamos 3 diversos lotes de tareas (*taskCPU* y *taskConsola*), los cuales fueron chequeados con 1,2 y 3 cores.
Trabajando con los mismos *quantum* cada core.

Nuestro primer lote de tareas fue el siguiente:

```
TaskCPU 70
TaskConsola 2 4 5
TaskCPU 40
TaskConsola 3 2 3
TaskCPU 30
```

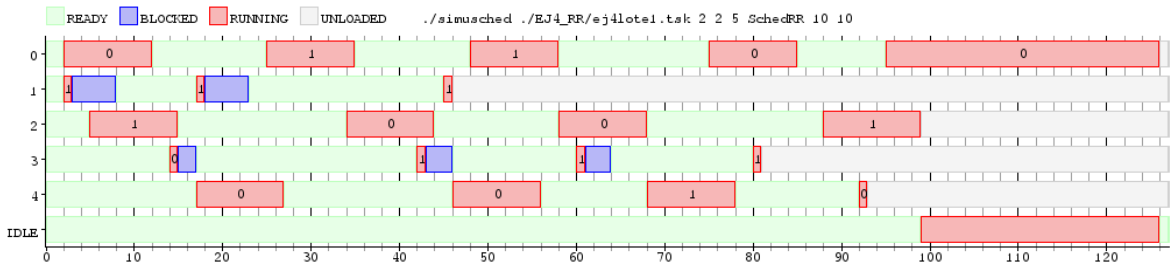
Utilizando este lote, obtuvimos los siguientes gráficos de simulación:



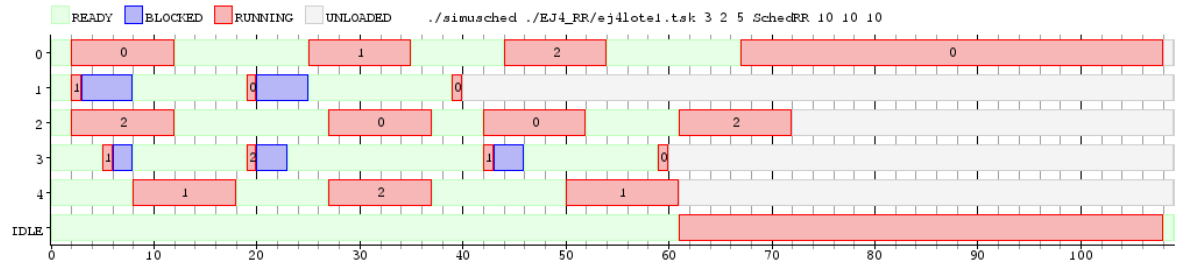
*Lote*1 - Scheduler RR - 1 core

Con esta simulación, trabajamos con 2 ticks de cambio de contexto y 5 de cambio de procesador, el cual no presta importancia en la simulación trabajando con un core.

Se puede observar el cambio de tareas cíclico tanto porque terminaron su quantum o porque se bloquearon.



*Lote*1 - Scheduler RR - 2 core



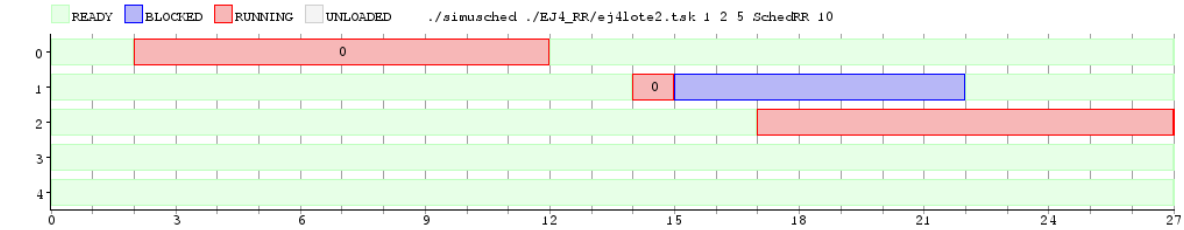
*Lote*1 - Scheduler RR - 3 core

Se ha podido notar, ademas del la ejecucion circular de las tareas, un cierto paralelismo al estar trabajando con 2 o 3 cores.

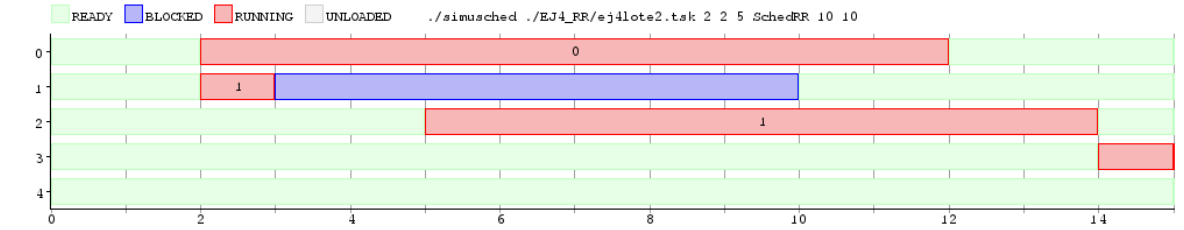
Luego, de esta simulación probamos con un lote con tareas que se bloqueen por más tiempo:

```
TaskCPU 70
TaskConsola 5 6 7
TaskCPU 40
TaskConsola 10 9 8
TaskCPU 30
```

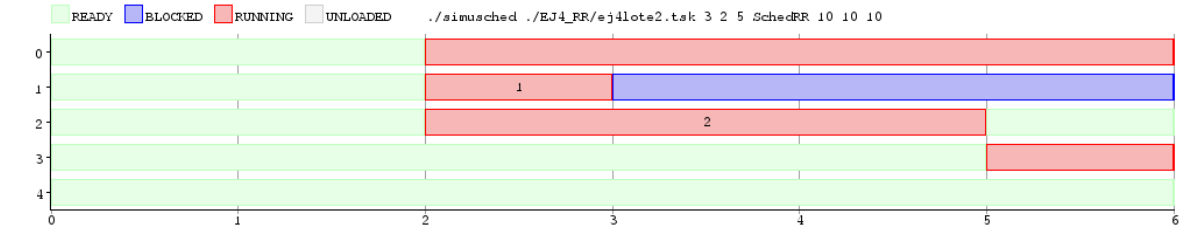
Manteniendo la misma cantidad de tick para cambio de contexto y core. Y manteniendo los mismos valores de *quantum* obtuvimos los siguientes gráficos:



*Lote*2 - Scheduler RR - 1 core



*Lote*2 - Scheduler RR - 2 core



*Lote*2 - Scheduler RR - 3 core

Con este lote, además de lo observado anteriormente pudimos ver que, al tener una tarea bloqueada por un largo tiempo, el scheduler directamente la ignora.

Luego de los gráficos pudimos observar lo siguiente sobre el comportamiento del Round-Robin:

- Carácter circular del algoritmo.
- Desalojo de las tareas cuando se bloquean o terminan y la inmediata asignación del núcleo a la siguiente tarea en caso de existir alguna.
- Libre de inanición.
- Una tarea bloqueada es ignorada por el scheduler hasta que se desbloquee.

Finalmente, dado su carácter circular y equitativo, podemos afirmar que todas las tareas que estén en condiciones de correr serán ejecutadas y ninguna será negada de tiempo de procesamiento.

4.2.3 Ejercicio 5

Lottery Scheduling

Lottery Scheduling se trata de un mecanismo que asigna recursos de manera aleatoria. Dando lugar a que cada proceso tenga una probabilidad p de ejecutarse.
Esta probabilidad de ganar un recurso, está representada por billetes de lotería (o tickets). A la hora de asignar un recurso (procesador) a un proceso se celebra una lotería, en la que un recurso es asignado al proceso con el ticket ganador.
Esto permite asignar los recursos eficaz y proporcionalmente.

Implementación de la idea básica del algoritmo

Para llevar a cabo este concepto, se implementó la idea de asignación de tickets por proceso. Es decir, que a cada proceso nuevo que llegue, el scheduler le asigna una cantidad determinada de tickets.
Esta cantidad de tickets en principio es igual para todos los procesos.
Por eso a la hora de cargarse un nuevo proceso, se recalcula la cantidad total de tickets.

El cálculo de la cantidad total de tickets y de los tickets que le corresponden a un proceso nuevo es el siguiente:

- tickets_totales = QUANTUM $*$ cantidad_de_procesos $*$ 1000;
- tickets_de_un_proceso = tickets_totales/cantidad_de_procesos;

Luego de esta asignación de tickets, la idea del algoritmo es sencilla.
Se elige un ticket ganador mediante la función rand entre 0 y la cantidad total de tickets menos uno. Para esto, previamente ya habíamos configurado (al construirse el scheduler) la semilla del generador de numeros aleatorios pasada por parámetro con la función srand.
Una vez conseguido el ticket ganador, hay un proceso ganador del quantum, y debemos buscarlo en la lista de procesos.
El quantum es un parámetro que usamos para que el proceso pueda correr más de un tick de reloj. El proceso ganador puede no llegar a terminar su quantum, ya que se bloqueó.
En caso de bloquearse el proceso, se lo saca de la lista de procesos disponibles, para que el scheduler no lo tenga en cuenta hasta que el mismo se desbloquee.
Al desbloquearse, el scheduler simplemente lo agrega de nuevo a la lista de procesos disponibles con la misma cantidad de tickets de un proceso nuevo.

Entonces, se puede afirmar, que todos los procesos tienen la misma probabilidad de ser elegidos por el scheduler.

Optimización de tickets compensatorios

La idea de esta opimización es sencilla. Se trata de darle más probabilidad de ganar el quantum al proceso que fue bloqueado. Cuando un proceso se bloquea, no terminó todo el quantum que había ganado.
Sabiendo esto, guardamos el pid del proceso a bloquearse en una lista de bloqueados, y también guardamos en otra lista el quantum que consumió.
Al desbloquearse, en vez de agregarlo una vez a la lista de procesos disponibles, lo agregamos la cantidad de veces como ticks le faltaba para terminar el quantum antes de bloquearse.
Por ejemplo, si el quantum es de 10 y consumió solo 4, lo agregamos 4 veces a la lista de procesos. Y aca está el punto clave de la optimización: el proceso bloqueado en realidad tiene más tickets que un proceso que no se bloqueó, en el ejemplo, 4 veces más. Por lo tanto tiene más posibilidades de volver a ser elegido.

5 Parte 3: Evaluando los algoritmos de scheduling

5.1 Ejercicios

- **Ejercicio 6** Programar un tipo de tarea TaskBatch que reciba dos parametros: total cpu y cant bloqueos. Una tarea de este tipo debera realizar cant bloqueos llamadas bloqueantes, en momentos elegidos pseudoaleatoriamente. En cada tal ocasion, la tarea debera permanecer bloqueada durante exactamente un (1) ciclo de reloj. El tiempo de CPU total que utilice una tarea TaskBatch debera ser de total cpu ciclos de reloj (incluyendo el tiempo utilizado para lanzar las llamadas bloqueantes; no ası el tiempo en que la tarea permanezca bloqueada).

- **Ejercicio 7** Elegir al menos dos metricas diferentes, definirlas y explicar la semantica de su definicion. Diseñar un lote de tareas TaskBatch, todas ellas con igual uso de CPU, pero con diversas cantidades de bloqueos. Simular este lote utilizando el algoritmo SchedRR y una variedad apropiada de valores de quantum. Mantener fijo en un (1) ciclo de reloj el costo de cambio de contexto y dos (2) ciclos el de migracion. Deben variar la cantidad de nucleos de procesamiento. Para cada una de las metricas elegidas, concluir cual es el valor optimo de quantum a los efectos de dicha metrica.

- **Ejercicio 8** Implemente un scheduler Round-Robin que no permita la migracion de procesos entre nucleos (SchedRR2). La asignacion de CPU se debe realizar en el momento en que se produce la carga de un proceso (load). El nucleo correspondiente a un nuevo proceso sera aquel con menor cantidad de procesos activos totales (RUNNING + BLOCKED + READY). Diseñe y realice un conjunto de experimentos que permita evaluar comparativamente las dos implementaciones de Round-Robin.

- **Ejercicio 9** Diseñar y llevar a cabo un experimento que permita poner a prueba la ecuanimidad (fairness) del algoritmo SchedLottery implementado. Tener en cuenta que, debido al factor pseudoaleatorio involucrado, cualquier corrida puntual podrıa ser arbitrariamente injusta; sin embargo, si se repite un mismo experimento n veces y se observan los resultados acumulativos, tales anomalıas deberıan ir desapareciendo conforme n aumenta. En otras palabras, interesa mostrar en base a evidencia empırica que el algoritmo implementado efectivamente tiende a ser totalmente ecuanime a medida que n tiende a infinito.

- **Ejercicio 10** Los autores del artıculo sobre lottery scheduling alegan que la optimizacion de compensation tickets es necesaria para compensar una posible falencia del algoritmo inicial- mente propuesto en ciertos escenarios. Diseñar y llevar a cabo un experimento apropiado para comprobar esta afirmacion (provocar un escenario donde se manifieste el problema, comparar simulaciones ejecutadas con y sin compensation tickets y discutir los resultados obtenidos).

5.2 Resultados y Conclusiones

5.2.1 Ejercicio 6

Al igual que con la tarea TaskConsola, mencionaremos nuestro implementación y acontinuación de la misma explicaremos ciertos puntos de la misma.

```
void TaskBatch(int pid, vector<int> params) {
     int total_cpu = params[0];
     int cant_bloqueos = params[1];
     srand(time(NULL));
     vector<bool> uso = vector<bool>(total_cpu);
     for(int i=0;i<(int)uso.size();i++)
        uso[i] = false;
for(int i=0;i<cant_bloqueos;i++) {
            int j = rand()%(uso.size());
            if(!uso[j])
               uso[j] = true;
            else
               i--;
```

```
        }
for(int i=0;i<(int)uso.size();i++) {
    if( uso[i] )
        uso_IO(pid,1);
    else
        uso_CPU(pid, 1);
   }
}
```

Para este tipo de tarea, creamos un vector de tamaño igual a $total_cpu$ el cual tendra bool, ya sea true o false dependiendo del uso que se le de dentro de la tarea, ya sea uso_IO o uso_CPU. En caso de ser uso_IO sera true, y sino false.
Luego, utilizaremos un ciclo que ira desde 0 hasta el tamaño del vector y dependiendo el valor booleano, usará la funciones dadas por la catedra uso_IO o uso_CPU.

5.2.2 Ejercicio 7

Las métricas elegidas fueron:

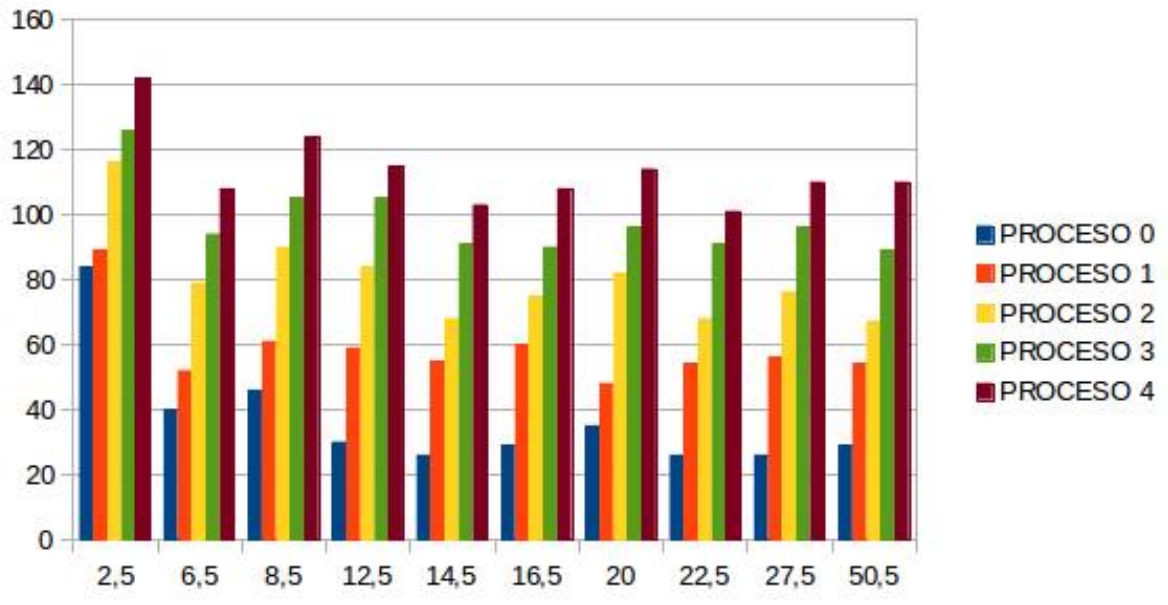
- **Turnaround**: Es el intervalo de tiempo desde que un proceso es cargado hasta que este finaliza su ejecución.
- **Waiting Time**: Es la suma de los intervalos de tiempo que un proceso estuvo en la cola de procesos *ready*.

Como las tareas TaskBatch se bloquean pseudoaleatoriamente, para obtener datos relevantes tomamos un promedio de las mediciones.

A la hora de encarar la experimentación, lo que realizamos fue simular corridas con varios quantum para poder obtener una aproximación del efecto del *quantum* en la ejecución del lote de tareas.

De esta aproximación, se confeccionaron gráficos de turnaround time en función del quantum, referente al estudio con 2 y 3 núcleos, los cuales se proveerán a continuación:

Luego de realizar mediciones con distintos *quantum*, tomamos la decisión de trabajar con los mismos *quantum* para cada nucleo al trabajar con más de 1 core. (Se muestra acontinuación un gráfico para demostrar que no era una decisión acertada trabajar con disversos *quantum* para cada core).



Turnaround - 2 core - Prueba Quantum distintos por Core

$$EjeX = Quantum;$$
$$EjeY = Tiempo$$

Al realizar, este gráfico y varias mediciones con distintos *quantum* por core, nos dimos cuenta que no terminaban siendo mediciones rigurosas, ya que una tarea podia estar corriendo con distintos tiempos por la migracion de procesos por core.

Por ende, optamos por realizar mediciones con igualdad de *quantum* por core para obtener la mejor performance posible.

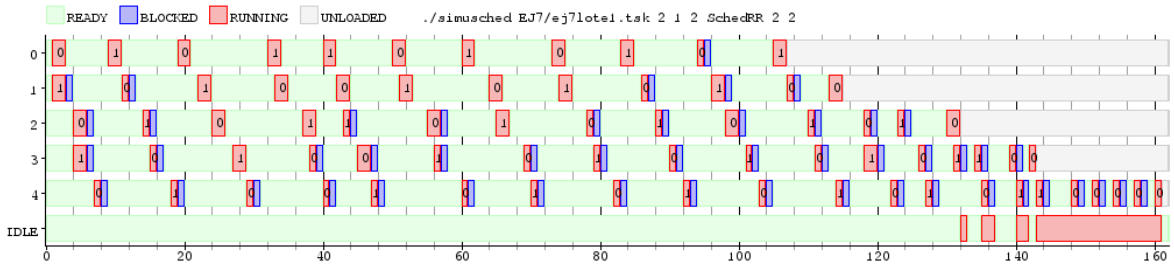
Por consiguiente, al trabajar con las nuevas mediciones, conjeturamos las siguientes hipótesis:

- Con 2 nucleos las mediciones de tiempo tienden a estabilizarse a partir de un *quantum* igual a 9.
- Con 3 nucleos las mediciones de tiempo tienden a estabilizarse a partir de un *quantum* igual a 11.

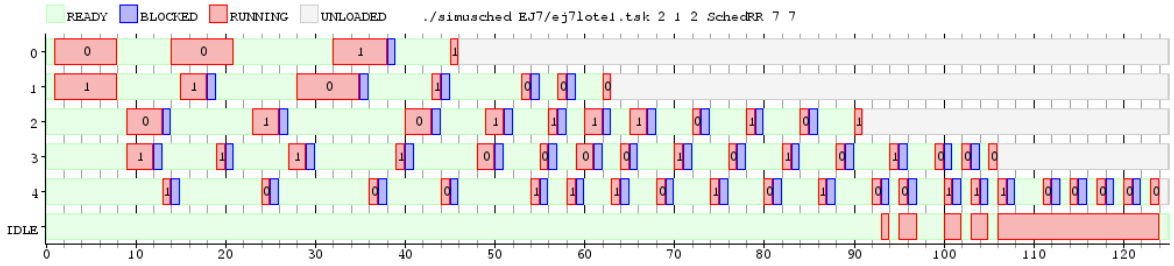
Turnaround Time

2 Core

A continuación se muestran los Diagramas de Gantt más relevantes:

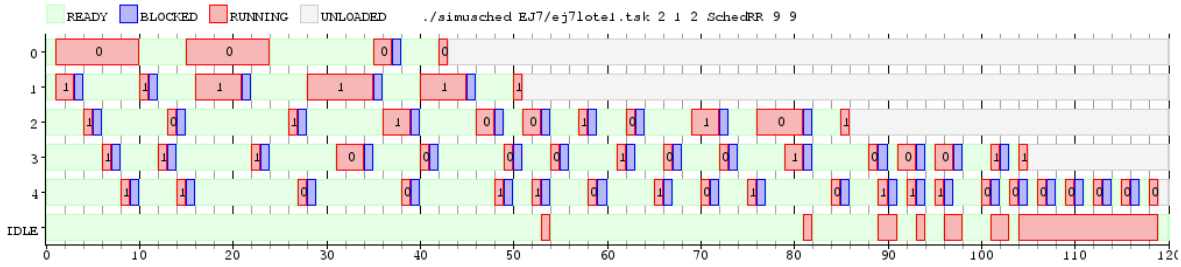


*Lote*1 - Turnaround - 2 core - Quantum igual a 2



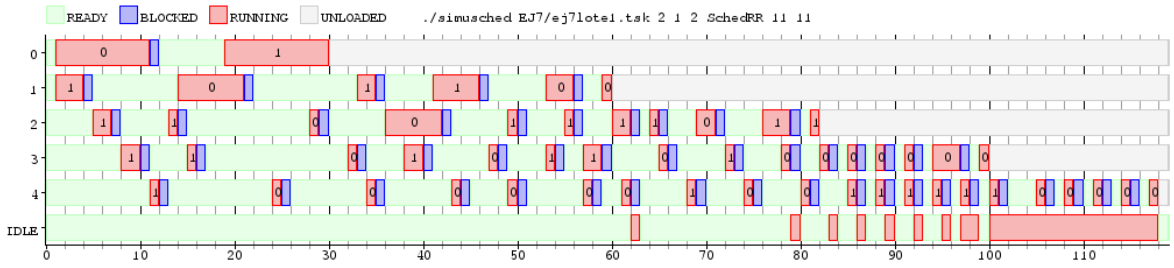
*Lote*1 - Turnaround - 2 core - Quantum igual a 7

La performance empieza a mejorar a medida que el *quantum* aumenta.

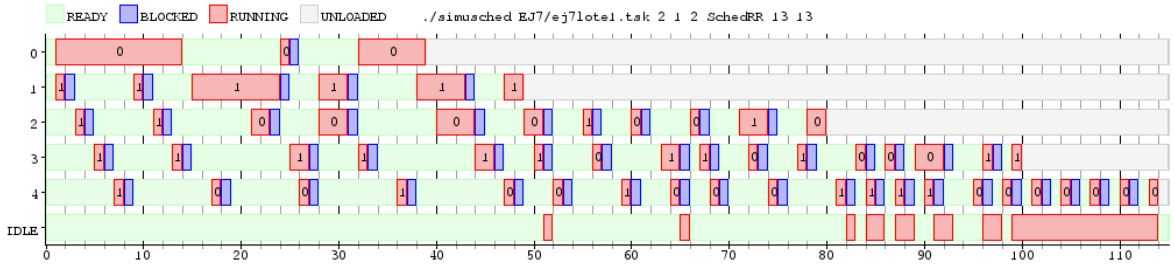


*Lote*1 - Turnaround - 2 core - Quantum igual a 9

La performance sigue mejorando,a partir de este valor, el desempeño comienza a estabilizarse, como muestran los siguiente dos gráficos. Las pequeñas diferencias en los valores responden a la pseudoaleatoriedad de las tareas TaskBatch.

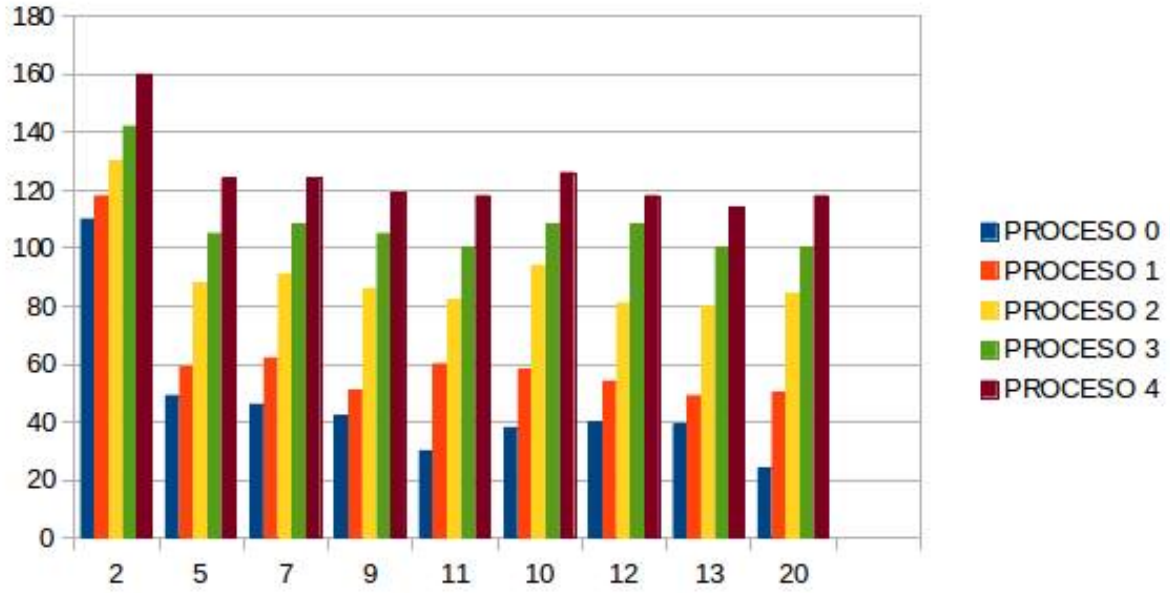


*Lote*1 - Turnaround - 2 core - Quantum igual a 11



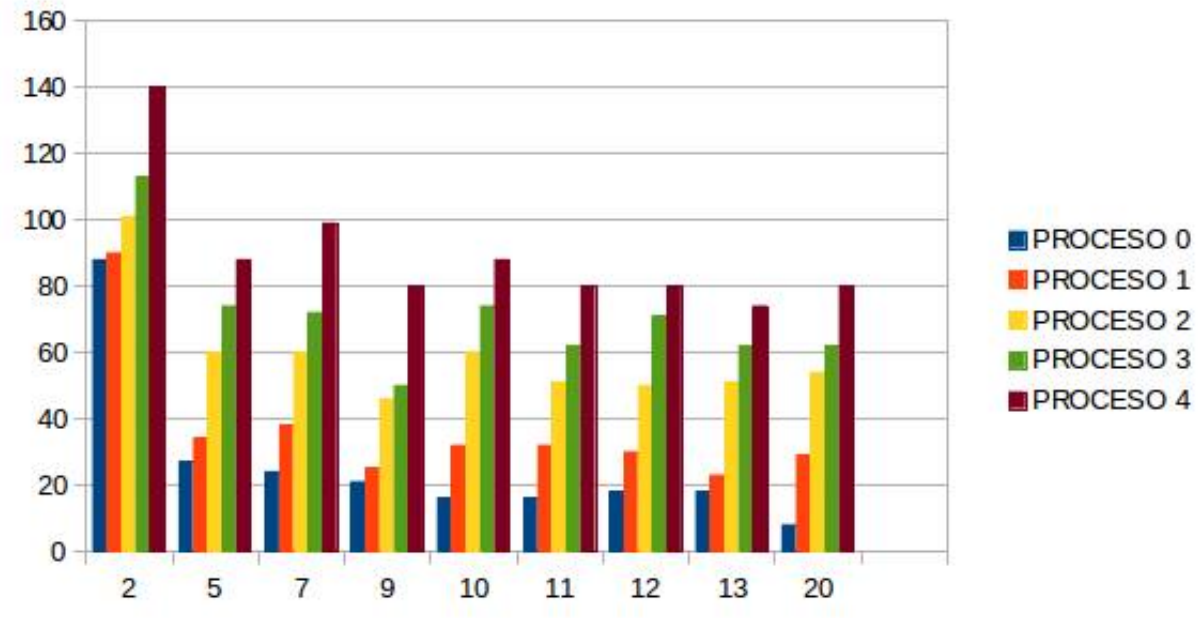
*Lote*1 - Turnaround - 2 core - Quantum igual a 13

Como mencionamos, las mediciones tienden a estabilizarse con un *quantum* igual a 9, pero la mejor performance obtenida es con un *quantum* igual a 13, teniendo en cuenta la pseudoaleatoridad del tipo de tarea utilizada. Se puede observar en la primer figura que a pesar de trabajar con 2 cores, al tener un quantum bajo (igual a 2) el costo es alto.



Turnaround - 2 core
$EjeX = Quantum$
$EjeY = Tiempo$

Waiting Time

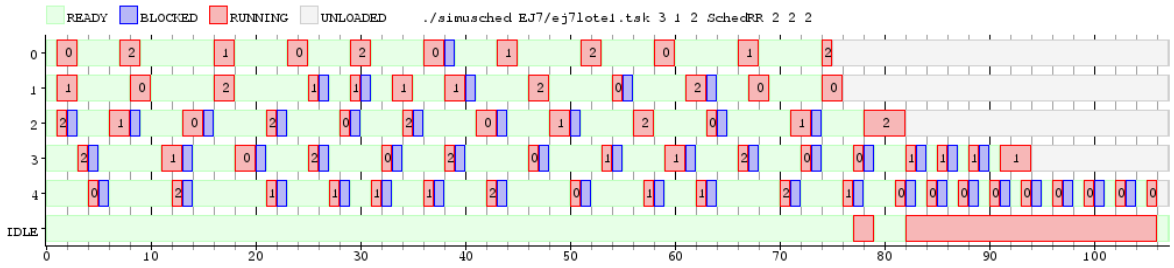


Waiting Time - 2 core
$EjeX = Quantum$
$EjeY = Tiempo$

Con este tipo de metrica, se comienza a estabilizar a partir del *quantum* igual a 5, obteniendo su mejor performance con el *quantum* igual a 13.

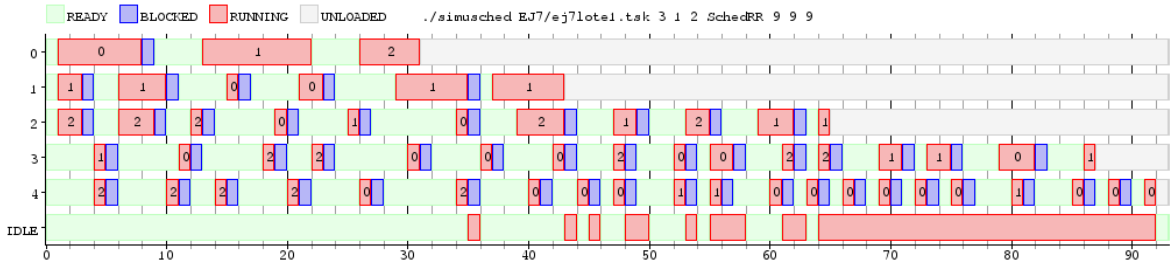
3 Core

A continuación, al igual que con 2 cores, mostraremos los Diagramas de Gantt mas relevantes:



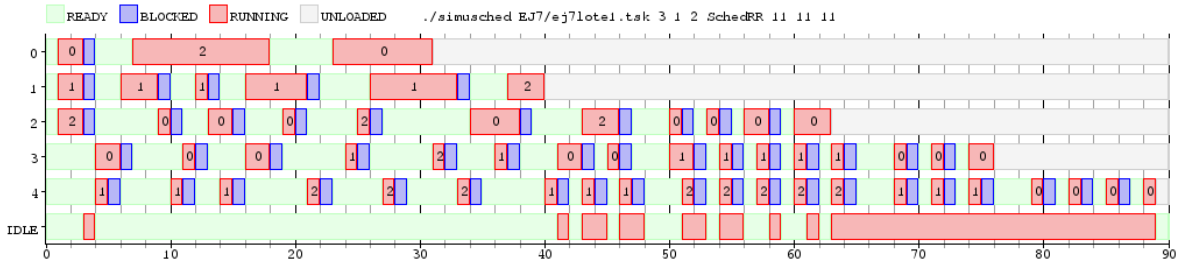
*Lote*1 - Turnaround - 3 core - Quantum igual a 2

Se observa que al tener otro core mas,a diferencia de con 2, a pesar de estar con un *quantum* bajo, la performance mejora bastante.

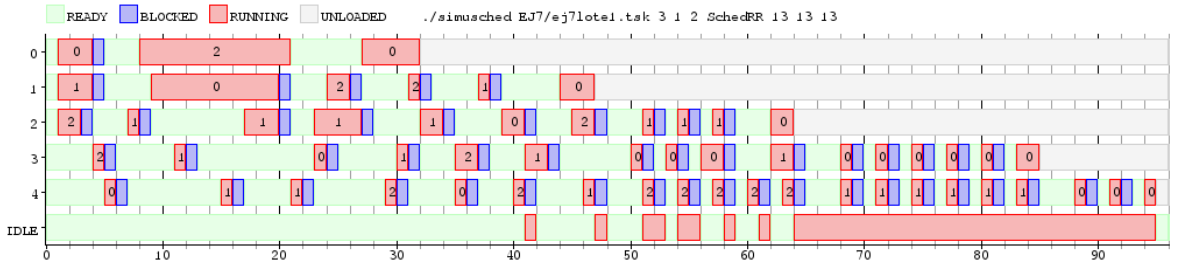


*Lote*1 - Turnaround - 3 core - Quantum igual a 9

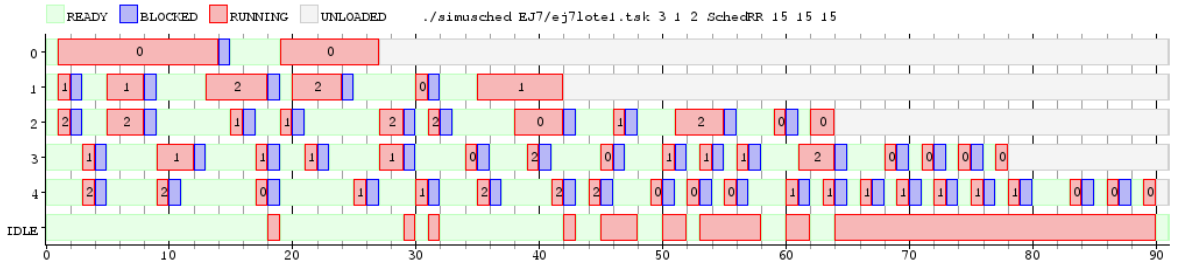
La performance empieza a mejorar a medida que el *quantum* aumenta.



*Lote*1 - Turnaround - 3 core - Quantum igual a 11

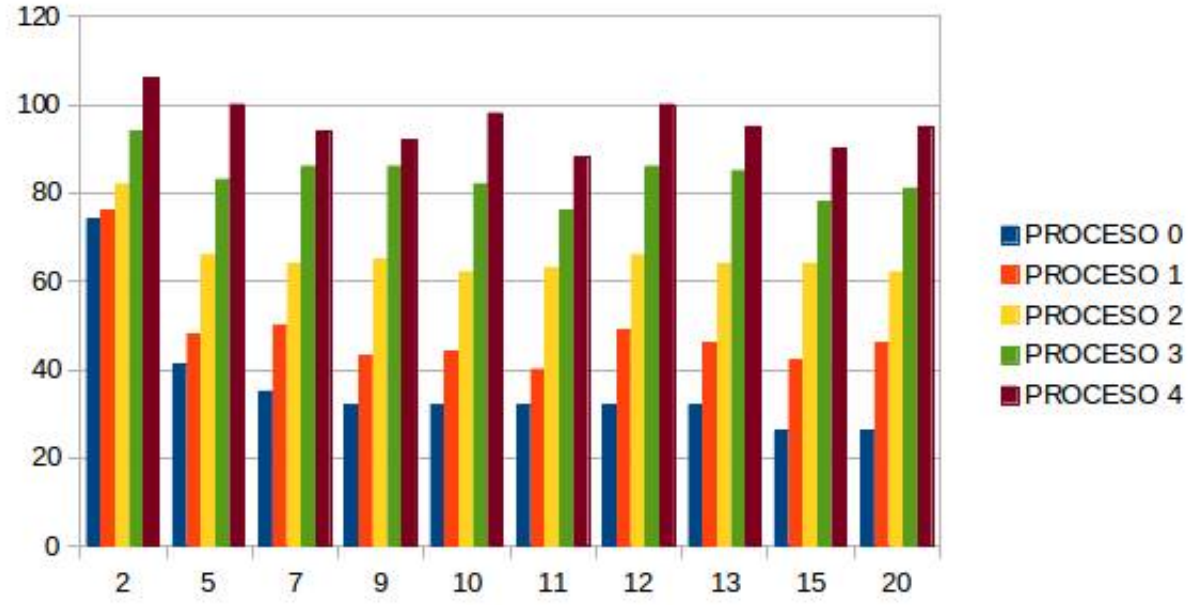


*Lote*1 - Turnaround - 3 core - Quantum igual a 13



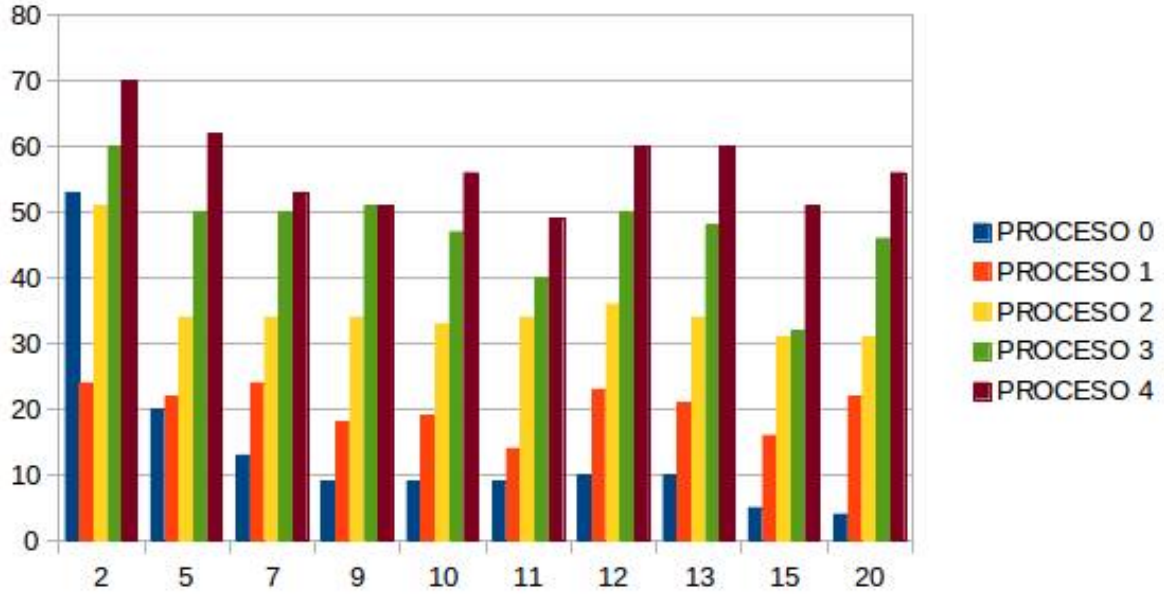
*Lote*1 - Turnaround - 3 core - Quantum igual a 15

A diferencia que en nuestra hipótesis conjeturada para con dos cores, con un *quantum* igual a 11 se obtiene la mejor performance, pero teniendo en cuenta la pseudoaleatoridad del tipo de tarea con la que se trabajo, a partir del *quantum* igual a 9 ya se estabiliza notoriamente.



Turnaround - 3 core
$EjeX = Quantum$
$EjeY = Tiempo$

Waiting Time



Waiting Time - 3 core
$EjeX = Quantum$
$EjeY = Tiempo$

Con este tipo de metrica,se obteniene a diferencia de con Turnaround su mejor performance con el *quantum* igual a 15.

Conclusiones

La diferencia entre los valores de quantum entre los casos se puede atribuir a que cada vez que agregamos un núcleo aumentamos la posibilidad de una migración de la tareas.

En todos lo casos se observa la influencia negativa que proviene de elegir un quantum con valores pequeños.

Agregar núcleos de procesamiento mejora significativamente la performance de acuerdo a la métricas con las que trabajamos, al permitir más procesamiento en pararelo y disminuyendo los waiting time de las tareas.

Fijada una cantidad de núcleos, aumentar el valor del quantum también mejora la performance, especialmente los tiempos referidos a las tareas que menos cantidad de bloqueos tienen. Igualmente, a partir de cierto valor de quantum, las mejoras en la performance dejan de ser muy significativas. Esto se produce a que las tareas con mas cantidad de bloqueos en algun momento dejan de consumir todo su quantum si seguimos aumentando el valor.

5.2.3 Ejercicio 8

La idea central de esta version de Round-Robin es que no permita migración entre núcleos y esto se basa en utilizar una cola FIFO por cada núcleo, donde se encolarán aquellas tareas a las que se asignó el core correspondiente.

Para implementar este algoritmo, el Round-Robin 2, utilizamos varias estructuras. Estas son:

- Los vectores *quantum* y *quantumActual*, que cumplen la misma función que en nuestra implementación de Round-Robin.

- El vector de colas *colas*, donde en la posición i se encontrará la cola de tareas correspondiente a ese núcleo de procesamiento.

- El diccionario *bloqueados*, donde mantenemos aquellas tareas que se bloquearon con su número de core correspondiente y que nos permitirá, cuando la tarea se desbloquee, reubicarla en la cola del core que le corresponde.

- El vector de enteros *cantidad*, cuya única función será tener en la posición i la cantidad de tareas bloqueadas, activas o en estado ready que están asignadas al core i y que nos servirá para determinar a qué núcleo se asignará una tarea al momento de cargarla.

Cuando se carga una tarea, se chequea cuál es el core que menor cantidad de procesos activos totales tiene asignados(haciendo uso de la posición respectiva del vector *cantidad*). Una vez que se obtiene dicho núcleo, se agrega la tarea a la cola de tareas correspondiente al core y se actualiza la cantidad de tareas activas para ese núcleo sumándole uno.

Al bloquearse una tarea, se define una entrada en el diccionario *bloqueados* con el pid y el núcleo correspondiente. De esta manera, al desbloquearse, obtenemos el core en el que debe correr, eliminamos la entrada del diccionario y encolamos nuevamente el pid a la cola del núcleo en cuestión. De esta manera resolvemos parcialmente el problema de no permitir la migración entre cores.

Finalmente, cuando una tarea termina, se actualiza la cantidad correspondiente al núcleo restándole uno. Solamente a la hora de cargar la tarea y cuando una tarea termina se modifica dicha variable. De esta manera, aunque una tarea se bloquee seguirá reflejada en la cantidad del núcleo, lo que nos permitirá seguir el criterio que nos pidieron en la consigna a la hora de asignarle un core a la tarea.

Como esta modificación de Round-Robin no permite migración entre cores, conjeturamos ciertos puntos:

- Dados un mismo lote de tareas y una misma configuración del scheduler (tick de costo en cambio de contexto y quantum) un único núcleo de procesamiento, ambos algoritmos deben comportarse de la misma manera.
- Comportamiento más eficiente en el Round-Robin 2 en lotes de tareas que se bloquean una gran cantidad de veces, o en configuraciones con *quantums* pequeños. Esto surge de que dichos escenarioshacen más proclive al Round-Robin a impulsar más cambios de contexto, con la posible penalidad de darse un cambio de núcleo, algo que puede ser muy costoso.

- El Round Robin debería comportarse más eficientemente en situaciones en las cuales el Round-Robin 2 pierde la posibilidad de parelelismo. Por ejemplo, varias tareas quedaron asignadas a un núcleo mientras los demás están libres. Generando que varios nucleos queden ociosamente bastante tiempo.

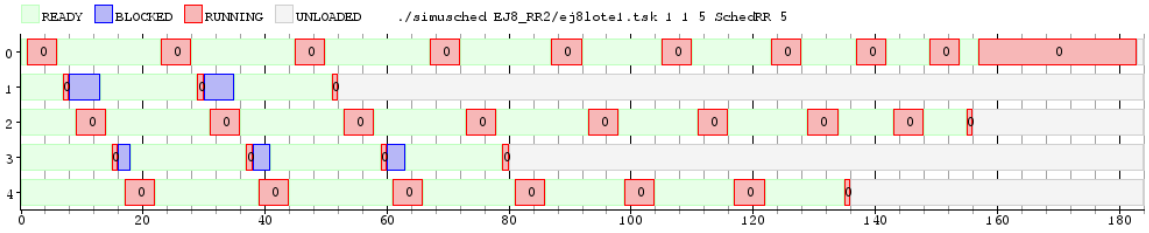
A continuación, mostraremos los experimentos diseñados para demostrar y explicar un poco mejor lo conjeturado:

Arrancando con nuestra primer conjetura:

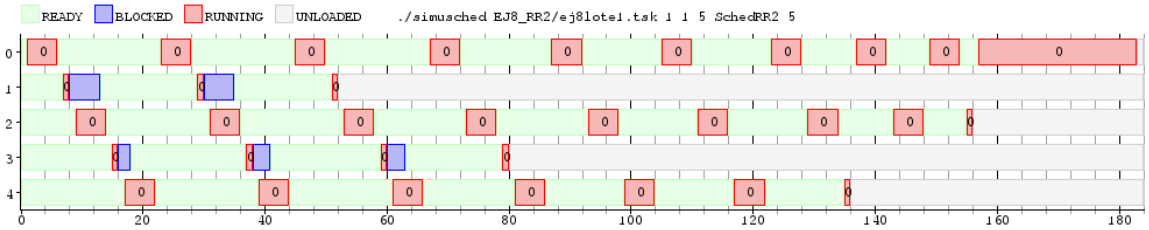
Dados un mismo lote de tareas y una misma configuración del scheduler con un único núcleo de procesamiento, ambos algoritmos deben comportarse de la misma manera.

Trabajando con el siguiente lote:

```
TaskCPU 70
TaskConsola 2 4 5
TaskCPU 40
TaskConsola 3 2 3
TaskCPU 30
```



*Lote*1 - Round Robin - 1 core - Quantum = 5 - cambio de contexto = 1

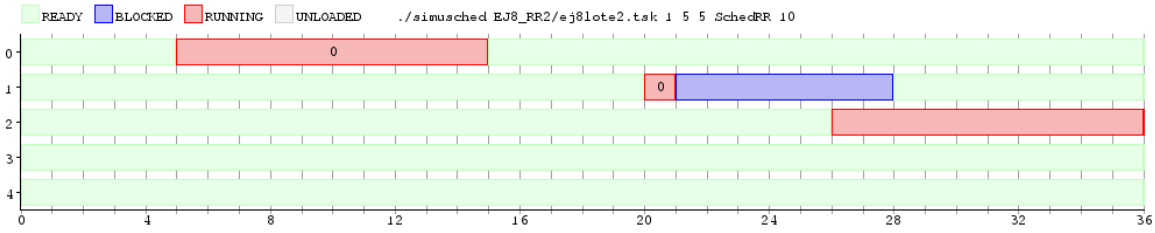


*Lote*1 - Round Robin 2 - 1 core - Quantum = 5 - cambio de contexto = 1

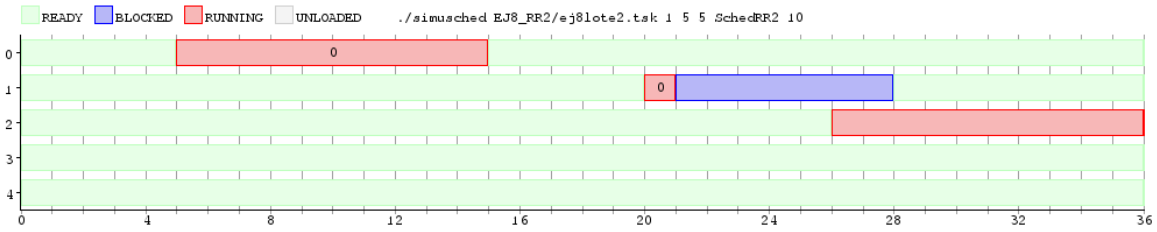
Se ve a simple vista, la igualdad entre ambos con este tipo de lote.

Ahora, con un lote distinto:

```
TaskCPU 70
TaskConsola 5 6 7
TaskCPU 40
TaskConsola 10 9 8
TaskCPU 30
```

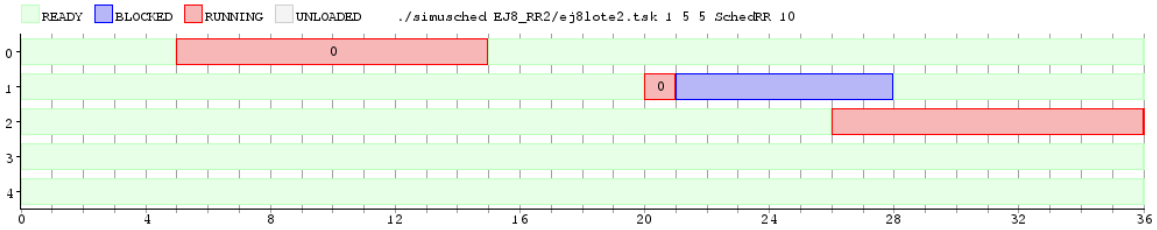


*Lote*2 - Round Robin - 1 core - Quantum = 10 - cambio de contexto = 5

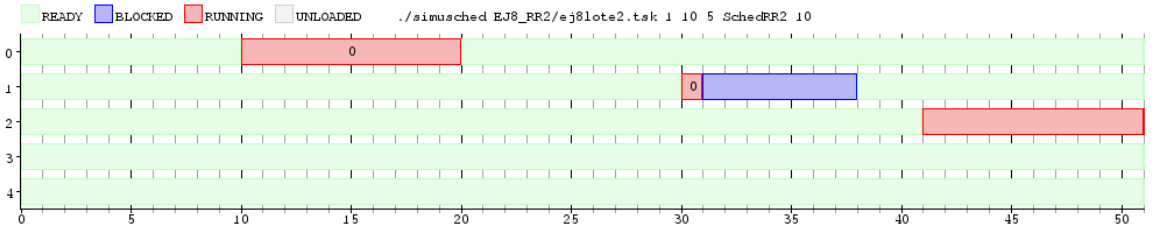


*Lote*2 - Round Robin 2 - 1 core - Quantum = 10 - cambio de contexto = 5

Nuevamente, observamos la misma igualdad entre ambos. De la unica manera en la cual se podria producir un cambio entre ambos Diagramas es alterando el cambio de contexto para uno y no para el otro.



*Lote*2 - Round Robin - 1 core - Quantum = 10 - cambio de contexto = 5



*Lote*2 - Round Robin 2 - 1 core - Quantum = 10 - cambio de contexto = 10

Aqui se ve las diferencias como lo mencionamos, por ende, podemos concluir que como el Round Robin 2 solo fue modificado para tener colas independientes para cada nucleo, al trabajar con 1 solo core, no se veran diferencias.

Continuamos con la siguiente conjetura realizada:

Comportamiento más eficiente en el Round-Robin 2 en lotes de tareas que se bloquean una gran cantidad de veces, o en configuraciones con *quantums* pequeños.

Arrancaremos mostrando casos con cambios de contexto y quantum pequeños:

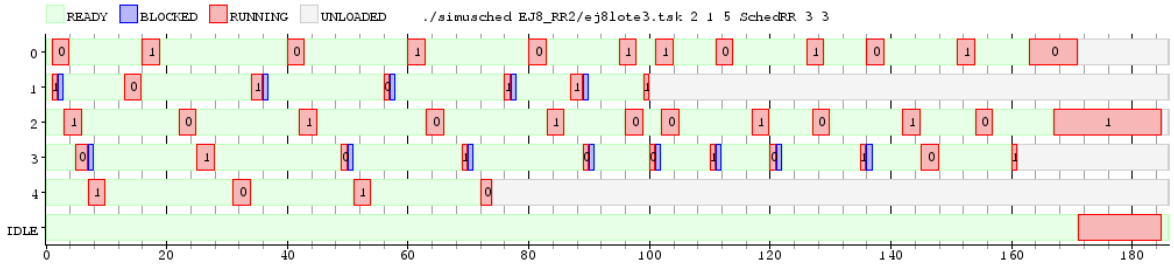
*Lote*1 - Round Robin - 2 core - Quantum = 3 - cambio de contexto = 1

*Lote*1 - Round Robin 2 - 2 core - Quantum = 3 - cambio de contexto = 1

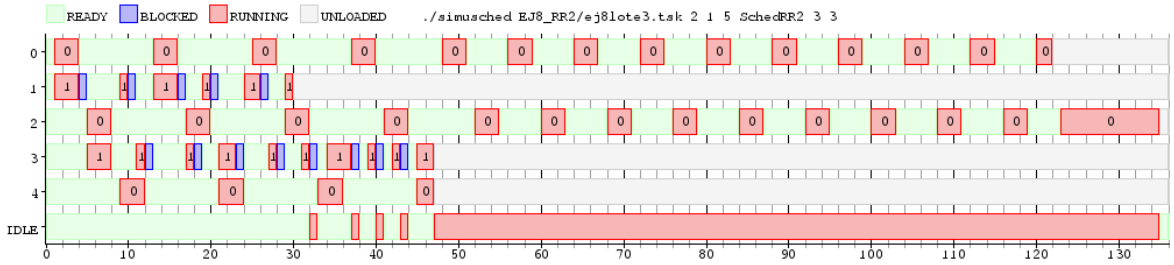
Se puede observar, una mejoria leve del Round Robin 2. Con la modificación realizada en el Round Robin 2, no habria ninguna modificacion en caso de aumentar o disminuir el cambio de migracion ya que este no lo permite.

Ahora, si trabajamos con tareas que utilicen el CPU y se bloqueen mucho mas podremos notar la mejor performance de este nuevo scheduler.

```
TaskCPU 40
TaskBatch 10  5
TaskCPU 50
TaskBatch 15 8
TaskCPU 10
```



*Lote*3 - Round Robin - 2 core - Quantum = 3 - cambio de contexto = 1



*Lote*3 - Round Robin 2 - 2 core - Quantum = 3 - cambio de contexto = 1

Se ve a simple vista, como el RR2 es mucho mejor, y termina aprox 50 milisegundos antes.
Como mencionamos anteriormente, si el cambio de migracion se alterase no tendria incidencia para este RR2 ya que no admite migracion de tareas, mientras que el anterior scheduler si, pudiendo empeorar a su vez su performance en caso de que el coso fuese mayor.

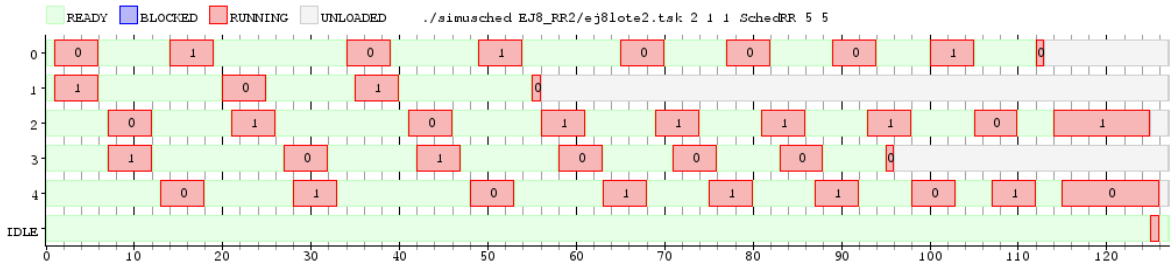
Por consiguiente, podemos afirmar, que nuestro RR2 sera notablemente mejor con tareas que utilicen el CPU y se bloqueen bastante tiempo.

Por ulimto, demostraremos nuestra ultima conjetura:

El Round Robin debería comportarse más eficientemente en situaciones en las cuales el Round-Robin 2 pierde la posibilidad de parelelismo.

Para probar esta conjetura, utilizamos tareas que demanden mas al CPU como $TaskCPU$.

```
TaskCPU 40
TaskCPU 15
TaskCPU 50
TaskCPU 30
TaskCPU 50
```



*Lote*3 - Round Robin - 2 core - Quantum = 5 - cambio de contexto = 1

*Lote*3 - Round Robin 2 - 2 core - Quantum = 5 - cambio de contexto = 1

Es notorio en estos gráficos cómo el Round-Robin, responde mejor que el Round-Robin 2. Esto es porque puede trabajar con procesamiento en paralelo, mientras que el Round-Robin 2 no, porque las 3 tareas quedaron asignadas a un core (el numero 0), aunque haya un core ocioso por un largo tiempo. En este caso, pagando el costo de migración se obtiene una mejor performance.

Con estos experimentos realizados logramos observar los comportamientos que habíamos conjeturado concluyendo que son ciertos.

Por ende, podemos concluir que el Round-Robin 2 responde de manera positiva para lotes con tareas de gran cantidad de bloqueos y de manera negativa al no poder paralelizar ciertos procesos.

5.2.4 Ejercicio 9

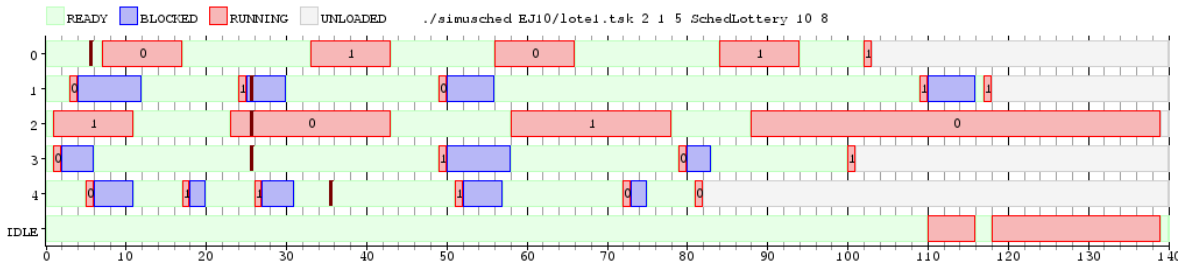
5.2.5 Ejercicio 10

Hemos encontrado, luego de varios experimentos, que este tipo de scheduler presenta graves problemas cuando se tienen varias tareas con mismo deadline.
Al no presentar ningun tipo de especificación, este scheduler, al no saber cuanto durará la tarea ejecuta cualquiera sin darle prioridad al orden de los deadline dado en el lote de tarea.
A continuación, expondremos un caso practico donde se puede ver como impacta dentro del desempeño del scheduler el orden en el que son cargadas las tareas pero la falta de informacion del scheduler imposibilita la posibilidad de cumplir la propiedad de carga respetando los deadline dados.

Uno de los lotes propuestos, fue el siguiente:

```
$5:
TaskCPU 40
$25:
TaskConsola 2 5 8
$25:
TaskCPU 100
$25:
TaskConsola 1 1 5
```

Donde el resultado que obtuvimos fue el siguiente:



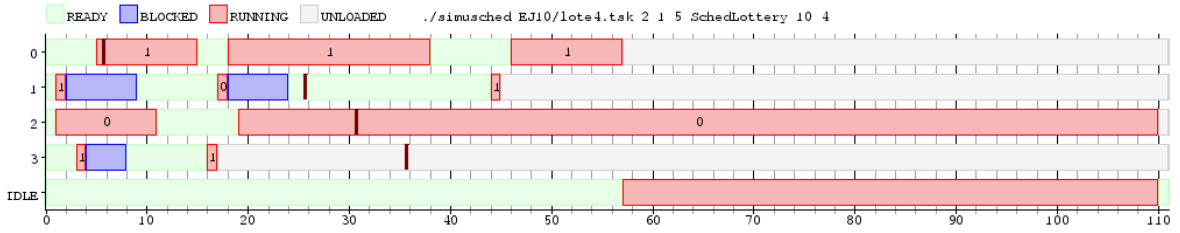
*Lote*2 - Lottery - 2 core

Como se puede observar, tanto el proceso 2 como 3 y 4 presentan el mismo deadline y terminan corriendo fuera del orden acordado a pesar de estar trabajando con varios cores a la ves.

Tambien, desarrollamos experimentos para los casos en que los deadline no sean los mismos, y obtuvimos resultados similares.
Uno de los lotes fue el siguiente:

```
$5:
TaskCPU 40
$25:
TaskConsola 2 5 8
$30:
TaskCPU 100
$35:
TaskConsola 1 1 5
```

Donde el resultado fue:



*Lote*4 - Lottery - 2 core

Como vemos, a pesar de tener distintos valores de deadline no se respeta el orden en este scheduler.

6 Bibliografía

- Cátedra de Sistemas Operativos - Clases teóricas y prácticas (2º Cuatrimestre 2014)
- Facultad de Ingenieria Uruguay
  (*https* : *//eva.fing.edu.uy/pluginfile.php/*75120*/mod_resource/content/*1*/*6*−SO−Teo−Planificacion.pdf*)
- Operating Systems Concepts, Abraham Silberschatz & Peter B. Galvin