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 **3**

2 Desarrollo y Resultados **4**

3 Parte I – Entendiendo el simulador simusched **4**
3.1 Ejercicios 4
3.2 Resultados y Conclusiones 4
3.2.1 Resolución Ejercicio 1 4
3.2.2 Resolución Ejercicio 2 4

4 Parte II: Extendiendo el simulador con nuevos schedulers **6**
4.1 Ejercicios 6
4.2 Resultados y Conclusiones 6
4.2.1 Resolución Ejercicio 3 6
4.2.2 Resolución Ejercicio 4 7
4.2.3 Resolución Ejercicio 5 9

5 Parte 3: Evaluando los algoritmos de scheduling **12**
5.1 Ejercicios 12
5.2 Resultados y Conclusiones 12
5.2.1 Resolución Ejercicio 6 12
5.2.2 Resolución Ejercicio 7 13
5.2.3 Resolución Ejercicio 8 19
5.2.4 Resolución Ejercicio 9 23

6 Bibliografía **26**

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 variantes del algoritmo de scheduling Round-Robin, uno que permite migración de tareas entre núcleos y otro que no. Luego mediante varios experimentos intentaremos comparar ambos algoritmos.
Asimismo, basándonos en el paper "Scheduling algorithms for multiprogramming in a hard-real-time environment" implementaremos una versión del algoritmo para schedulers con prioridades dinámicas y estáticas en el que también 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 ximplementación, 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*. Esa función uso_IO simula una llamada bloqueante.

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

A continuacion, el codigo mencionado:

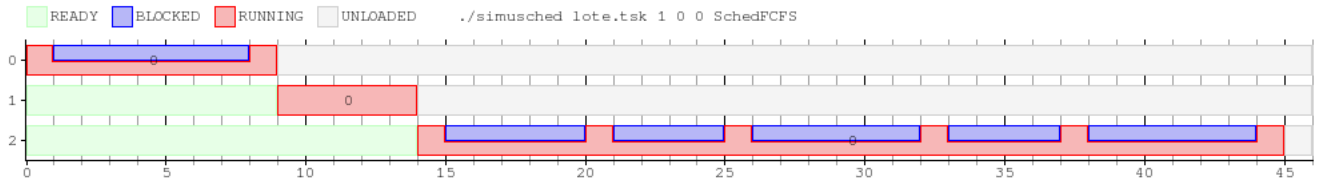
```
void TaskConsola(int pid, vector<int> params) {
     int i, ciclos;
     for (i = 0; i < params[0]; i++) {
            ciclos = rand() % (params[2] - params[1] + 1) + params[1];
            uso_IO(pid, ciclos);
     }
}
```

3.2.2 Ejercicio 2

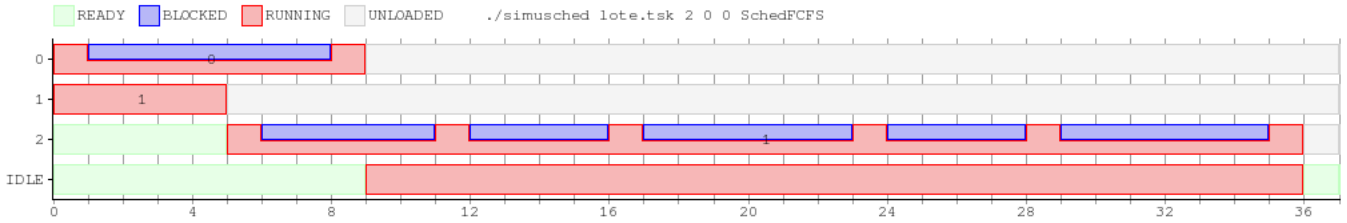
Para este punto, utilizamos el siguiente lote de tareas:

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

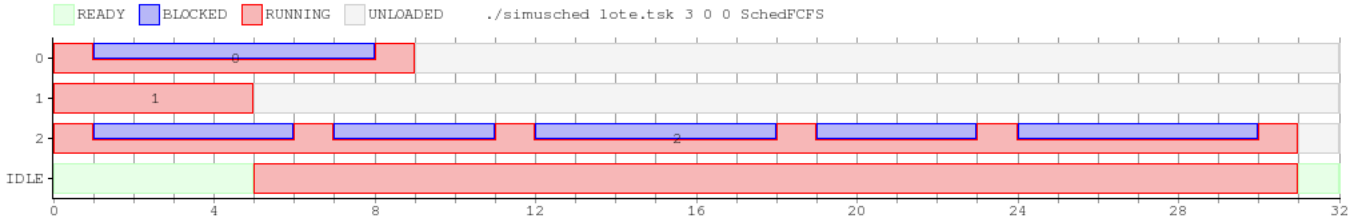
El mismo, presenta una tarea de uso intensivo $TaskCPU$ que dura unos 4 ticks, y otras dos interactivas, las cuales se bloquean 1 y 5 ticks respectivamente con una duración de entre 4 y 8 para la primera y 3 y 6 para la segunda.
A continuación, los respectivos gráficos de mediciones.



*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 núcleo 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 implementación 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 articulo:

  – 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.

  1. Responda:

     1. ¿Que problema estan intentando resolver los autores?
     2. ¿Por que introducen el algoritmo de la seccion 7? ¿Que problema buscan resolver con esto?
     3. Explicar coloquialmente el significado del teorema 7.

  2. Diseñar e implementar un scheduler basado en prioridades fijas y otro en prioridades dinamicas. Para eso complete las clases $SchedFixed$ y $SchedDynamic$ que se encuentran en los archivos $sched\_fixed.[h|cpp]$ y $sched\_dynamic.[h|cpp]$ respectivamente.

4.2 Resultados y Conclusiones

4.2.1 Ejercicio 3

Para desarrollar la implementación del scheduler $Round - Robin$ y que este funcione de una forma correcta utilizamos una serie de estructuras puntuales.
Las mismas son las siguientes:

1. Una cola global, la cual nombramos q, esta contiene los PID de los procesos activos que no estan bloqueados y en el tope de la misma se encuentra el próximo proceso a correr. Esta cola, fue desarrollada para que cuando se desaloje un proceso por finalizar su *quantum* la misma pase al final de la cola y generando el ciclo acorde al comportamiento de este scheduler.
2. Un vector denominado *cores*, este tiene en su elemento i el pid correspondiente a al proceso que está corriendo en el core $i + 1$. Inicializamos todos los elementos en -1, esto corresponde a la Idle Task, de esta forma reconocemos que no se cargaron procesos en los núcleos.
3. Un vector *quantum* guarda en la posicion i el quantum que se dispuso a cada núcleo.
4. Un vector *quantumActual* aqui guardaremos la cantidad de ticks que le quedan al proceso desde que fue cargado en el core.
5. Una lista de *bloqueados* esta tendra procesos que se bloquearon cuando estaban corriendo.

De esta manera, con estas estructuras nos permiten determinar para cada tarea, cuándo, y cuánto de su quantum consumieron de forma que podamos desalojarla correctamente.

A su vez, tomamos ciertas decisiones en esta implementación:

- 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. Además, sera reseteado el quantum, se le dará inicio a la próxima tarea que se encuentre ready y cuando el sistema operativo, nos envie una señal de unblock, la tarea desalojada regresará al final de la cola global.

4.2.2 Ejercicio 4

El algoritmo de scheduler **Round-Robin** tiene como característica asignar a todas las tareas un determinado tiempo máximo de procesamiento, a esto se lo llama *quantum*.

Este tiempo esta definido para cada núcleo en particular, dependiendo de en cuál de ellos estén ejecutando los procesos, se les asignará el respectivo tiempo máximo.

Otra característica del **Round-Robin** es que las tareas se encolan y se ejecutan cíclicamente. Osea que cuando se deja de ejecutar, si no terminó su ejecución, la tarea se encolará al final de la lista. Como elección de diseño, elegimos que se use una cola global para todos los procesadores, aunque también se podría tener una cola para cada núcleo.

A su vez, también puede ocurrir una tarea no consuma todo su *quantum*. Ya sea porque la tarea se bloquea (haciendo uso de dispositivos de entrada/salida) o porque termine su ejecución.

En caso de haber terminado, nuestro algoritmo pone a correr directamente la próxima tarea de acuerdo al orden circular que se estableció y la tarea que finalizó se desalojará por completo y no sera considerada nuevamente.

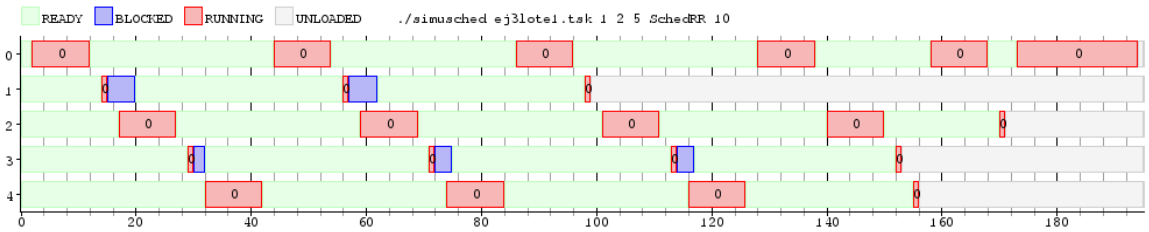
En caso de haberse bloqueado, esta misma dejará de ser considerada hasta que se desbloquee, perdiendo el quantum que le quedaba si hubiere. Automáticamente, seguirá corriendo la próxima tarea que se encuentre en la cola global. Cuando el proceso se desbloquee, será encolada nuevamente al final de dicha cola.

Para corroborar que el comportamiento era el deseado, desarrollamos 3 disversos lotes de tareas compuestos por tareas del tipo *taskConsola* y *taskCpu*, trabajando con 1, 2 y 3 cores y utilizando el mismo *quantum* para cada uno de los mismos.

Nuestro primer lote de tareas fue el siguiente:

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

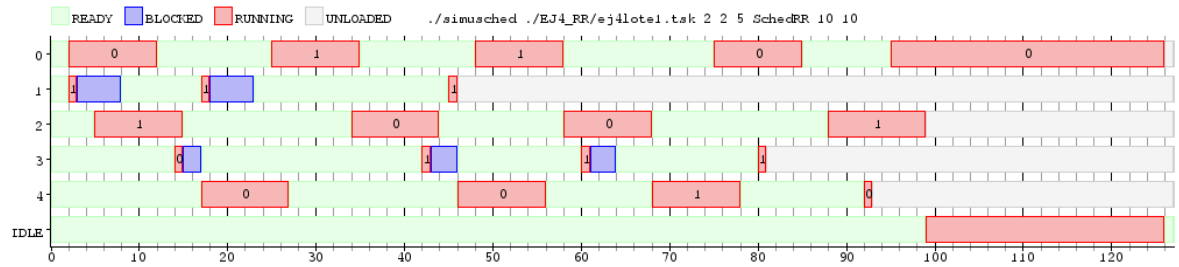
Obteniendo los siguientes resultados:



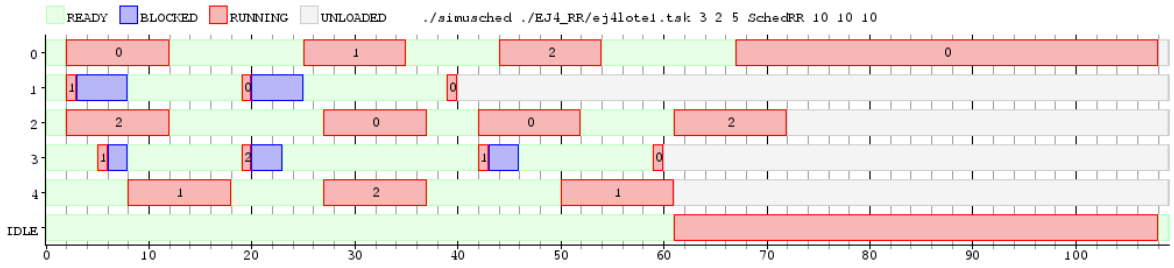
*Lote*1 - Scheduler RR - 1 core

Con esta simulación, trabajamos con 2 ticks de cambio de contexto.

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



*Lote*1 - Scheduler RR - 2 core



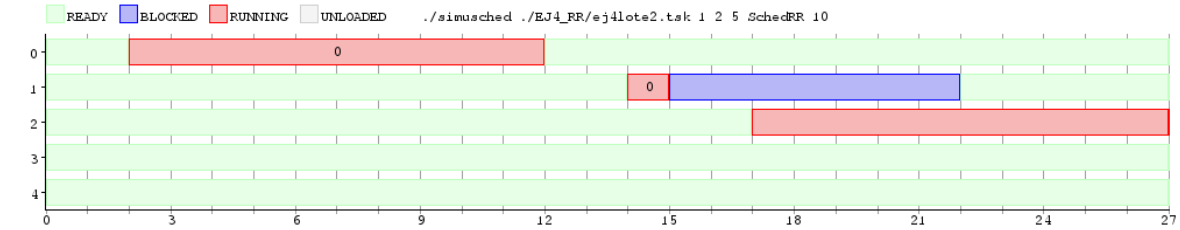
*Lote*1 - Scheduler RR - 3 core

Es notorio, ademas del la ejecuciòn circular de las tareas, un cierto paralelismo al estar trabajando con 2 o 3 cores.

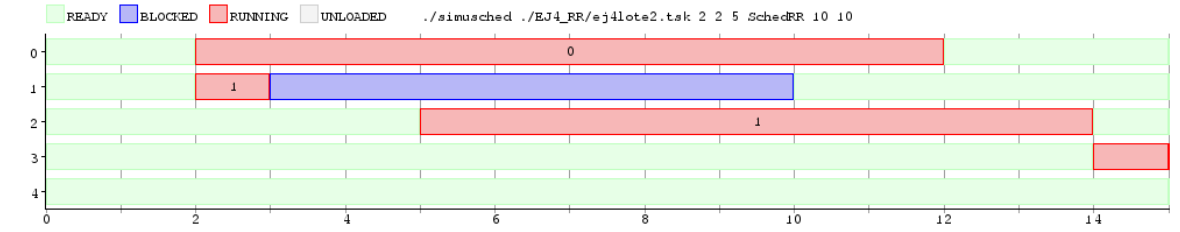
Luego, de esta simulación probamos con un lote con tareas que se bloqueen por más tiempo:

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

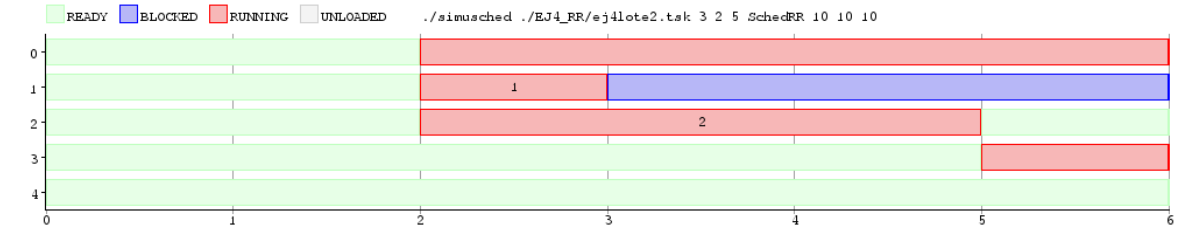
Manteniendo la misma cantidad de tick para cambio de contexto y core. También, nos parecio prudente, mantener los mismos valores de *quantum* obteniendo los siguientes gráficos:



*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 estos experimentos pudimos observar ciertos puntos del 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

PARTE 1

¿Qué problema están intentando resolver los autores?

En el paper "Scheduling algorithms for multiprogramming in a hard-real-time environment" los autores están intentando mostrar y resolver un problema que estaba surgiendo en esa época (1973). El uso de computadoras que controlaban y monitoreaban procesos industriales había empezado a incrementarse notablemente. Con esto aparecen los sistemas que tienen un "tiempo crítico" para procesar. Es decir, sistemas en donde el procesamiento estaba sujeto a un tiempo máximo al cual ejecutarse.
Por lo tanto necesitaban maximizar la eficiencia del uso del procesamiento, y para eso afirman que eso es posible mediante un algoritmo de schedulling que asegure procesar antes de ese tiempo crítico.
Por eso en el paper proponen dos algoritmos que manejan prioridades: prioridades fijas y dinámicas.

¿Por qué introducen el algoritmo de la sección 7?
¿Qué problema buscan resolver con esto?

En la sección 7 introducen el algoritmo de prioridades dinamicas tratando de resolver un problema que muestra el algoritmo de prioridades fijas descripto en las secciones anteriores.
Al tener prioridades fijas, puede ocurrir inanición. Es decir, puede pasar que siempre lleguen tareas con prioridades más altas de las que se están por corriendo o por correr, dejándolas en espera y nunca atendiéndolas.
El algoritmo que presenta en esta sección tiene el nombre de "The deadline driven scheduling algorithm", explicando que se trata de un algoritmo que da más prioridad a las tareas en las que su deadline esté mas cercano.

Explicar coloquialmente el significado del teorema 7

El teorema 7 enuncia lo siguiente:

For a given set of m task, the deadline driven scheduling algorithm is feasible if and only if

$$(C_1/T_1) + (C_2/T_2) + ... + (C_m/T_m) \leq 1$$

Dicho teorema habla sobre la factibilidad y uso de las tareas en el algoritmo que se utilice como solución a la particular dificultad del algoritmo del scheduler fixed.
Puntualmente el lote de tareas m deberá cumplir una precondición en donde la suma de cada una de las relaciones entre el run time o tiempo de ejecución de la tarea, sobre el período de la misma de todas las tareas sea igual o menor a 1.

Esto quiere decir que el tiempo de ejecución de una tarea deberá ser considerablemente menor al período de la misma para que el desarrollo y comportamiento del scheduler sea el deseado y pudiese resolver la dificultad que se había generado por el anterior.

PARTE 2

Algoritmo Scheduler Fixed - Explicación de Implementación

Luego del estudio del Paper solicitado se realizo la implementación del dicho algoritmo, utilizando una serie de estructuras para que este funcione acorde a lo pedido:

- Un struct denominado **tarea** el cual contiene el pid, el run_time_actual, periodo dandonos informacion de la tarea cargada.
- Una lista de *tarea* nombrada **tareas**, esta es ordenada segun el periodo de las tareas de menor a mayor, obteniendo la prioridad del algoritmo.
- Una variable global **primera_pasada** con la cual podemos realizar nuestra primer comparacion de periodos entre tareas.

Ademas de estas estructuras utilizamos una funcion privada **insertarOrdenado** la cual como el nombre lo dice va guardando en nuestra lista de tareas, las tareas a corde se van cargando en su respectivo lugar comparando los periodos de las mismas.
De esta forma, la secuencia del algoritmo fue la siguiente:

1. Llega una nueva tarea se la carga e inserta ordenadamente en la lista.
2. La tarea corre hasta finalizar.
3. Una vez que finaliza se chequea en la lista cual es la primera en orden de prioridad queda esta ready para correr.

De esta forma, con las estructuras y funciones logramos que nuestro scheduler trabaje de la forma deseada.

Vale aclarar que, para que el scheduler funcione correctamente, es recomendable que los lotes de tareas cumplan la factibilidad enunciada en el paper estudiado.

Algoritmo Scheduler Dynamic - Explicación de Implementación

Una vez finalizado el estudio del paper, y habiendo notado las dificultades puntuales que enuncia el mismo sobre el anterior Scheduler, se solicito desarrollar uno nuevo que solucione dichos problemas.

Para el desarrollo del mismo trabajamos con ciertas estructuras puntuales, las cuales enunciamos a continuación:

- Un struct denominado**tarea** el cual contiene el pid, el run_time_actual, periodo y el deadline para tener informacion de la tarea cargada.
- Una lista de *tarea* nombrada **tareas**, esta es ordenada segun el deadline de las tareas de menor a mayor, obteniendo la prioridad del algoritmo.
- Una variable global **primera_pasada** con la cual podemos realizar nuestra primer comparacion de periodos entre tareas.

Además de estas estructuras puntuales, definimos unas funciones de forma privada para poder desarrollar el Scheduler en cuestión:

- Función **insertarOrdenado**
- Función **chequearPeriodos**

La función **insertarOrdenado** como el nombre lo dice va guardando en nuestra lista *tareas* las tareas acorde van siendo cargadas respetando un orden, dicho orden se basa en la prioridad dada por los deadline de las tareas. A menor deadline la tarea quedara en los primeros lugares de la lista.

La función **chequearPeriodos**, la cual revisa si la tarea que se encuentra al principio de la lista tiene su deadline menor o igual a cero.

A partir de estas estructuras y funciones procedemos a explicar la secuencia de nuestro algoritmo:

1. Llega una nueva tarea, se chequea si es la primera en llegar y en caso de no serlo se la inserta ordenadamente.
2. Empieza a correr una tarea, y en cada Tick de reloj del cpu se va disminuyendo en uno el periodo y deadline de todas las demas tareas que no estan corriendo, mientras que la que se encuentra ejecutando se le disminuye en uno su run_time y su periodo.
3. En caso de que nuestra función booleana *chequearPeriodos* de True en un tick de reloj y la tarea que este corriendo no haya finalizado esta sera desalojada guardando su run_time actual y su deadline insertandola ordenadamente en la lista para que luego pueda finalizar su ejecución.
4. Si *chequearPeriodos* no dio nunca True durante la ejecución de la tarea, esta podra finalizar sin inconvenientes y posteriormente se cargará la primer tarea de la lista en orden de prioridad.

De esta forma, nuestro Scheduler continua hasta finalizar todas las tareas manteniendo esta secuencia.

Vale aclarar, como en el anterior algoritmo, que para un correcto funcionamiento de nuestro Scheduler es recomendable que el lote de tareas a utilizar cumpla la factibilidad enunciada en el paper.

5 Parte 3: Evaluando los algoritmos de scheduling

5.1 Ejercicios

- **Ejercicio 6** Programar un tipo de tarea TaskBatch que reciba dos parámetros: total cpu y cant bloqueos. Una tarea de este tipo deberá realizar cant bloqueos llamadas bloqueantes, en momentos elegidos pseudoaleatoriamente. En cada tal ocasión, la tarea deberá 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 métricas diferentes, definirlas y explicar la semántica de su definición. 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 migración. Deben variar la cantidad de nucleos de procesamiento. Para cada una de las métricas elegidas, concluir cual es el valor óptimo de quantum a los efectos de dicha métrica.

- **Ejercicio 8** Implemente un scheduler Round-Robin que no permita la migración de procesos entre nucleos (SchedRR2). La asignación 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** Disenar un lote de tareas cuyo scheduling no sea factible para el algoritmo de prioridades fijas pero sı para el algoritmo de prioridades dinámicas.

5.2 Resultados y Conclusiones

5.2.1 Ejercicio 6

Al igual que con la tarea TaskConsola, mencionaremos nuestro implementación y por consiguiente 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 tendrá 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 irá 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:

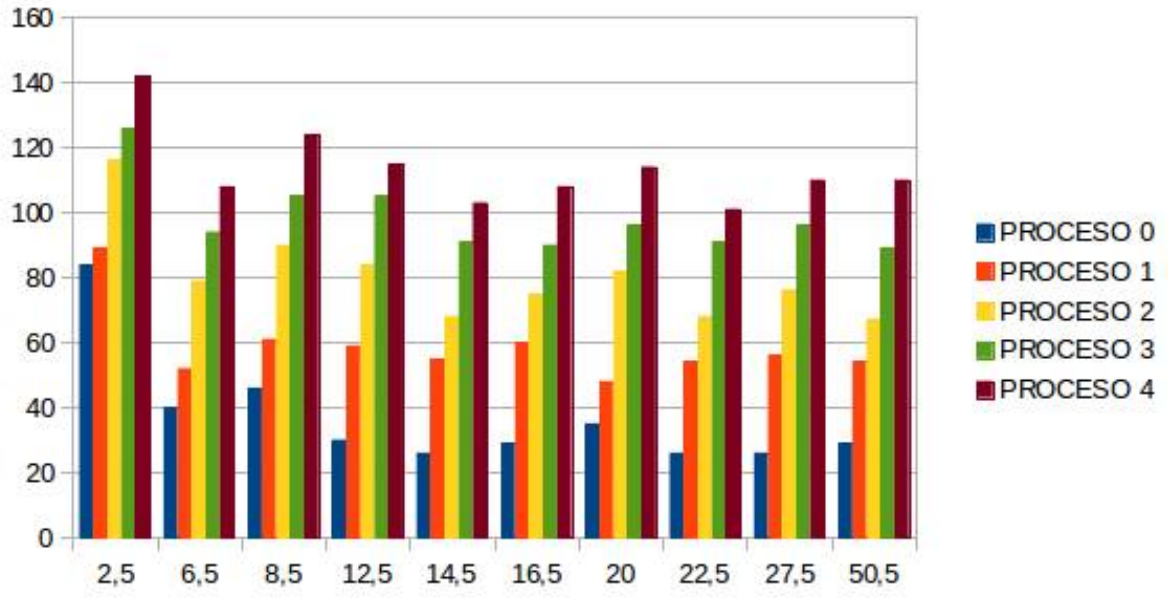
- **Turnaround**: Es el intervalo de tiempo desde que un proceso es cargado hasta que este finaliza su ejecución.
- **Waiting Time**: Es la suma de los intervalos de tiempo que un proceso estuvo en la cola de procesos *ready*.

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 hicimos fue trabajar con varios quantum distintos para poder obtener una mejor apreciación del efecto del *quantum* en la ejecución de este tipo de tareas.

A partir de este resultado, desarrollamos los siguientes gráficos de turnaround time en función del quantum, utilizando 2 y 3 núcleos

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. Igualmente, mostraremos a continuación un gráfico ejemplificando la diferencia que se produce al trabajar con distintos *quantums* por 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, observamos que no eran mediciones rigurosas, ya que un proceso podría estar corriendo con distintos tiempos por la migración de procesos por core.

Por consiguiente, al trabajar con las nuevas mediciones, concluimos con 2 hipótesis:

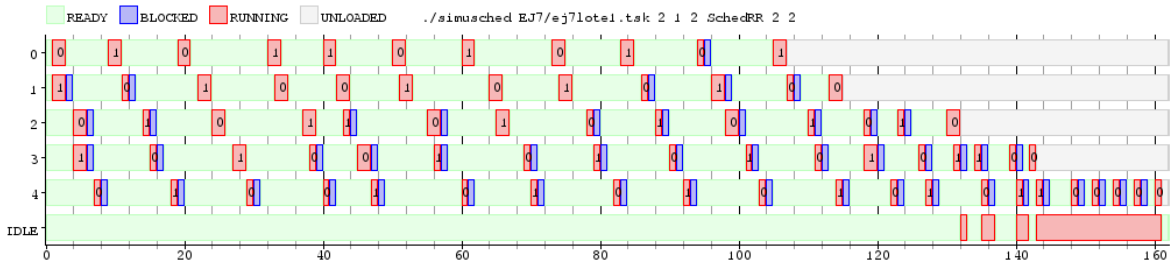
- Con 2 nucleos las mediciones de tiempo tienden a estabilizarse a partir de un *quantum* igual a 9.

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

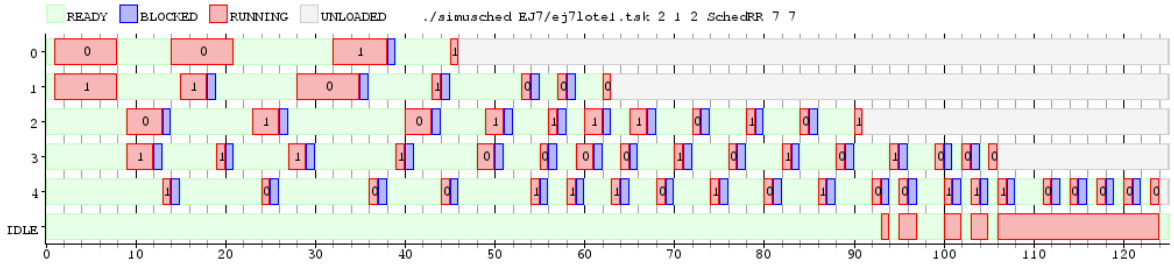
Turnaround Time

2 Core

A continuación se muestran los Diagramas de Gantt más relevantes:

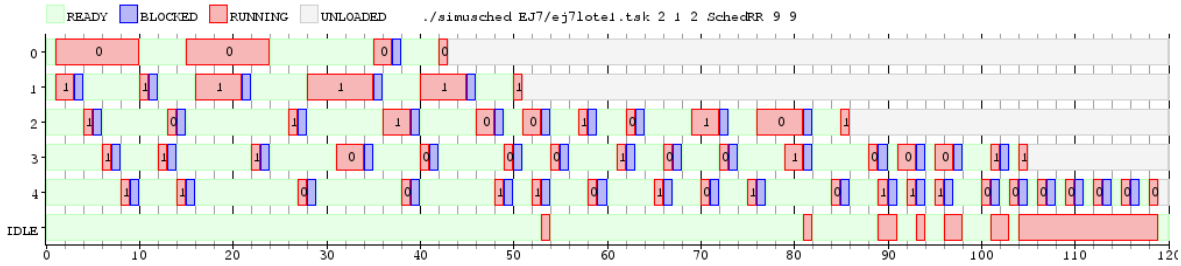


*Lote*1 - Turnaround - 2 core - Quantum igual a 2



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

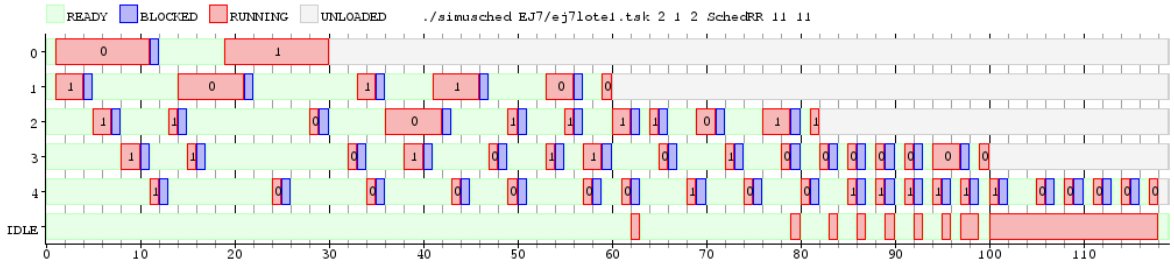
La performance empieza a mejorar a medida que el *quantum* aumenta.



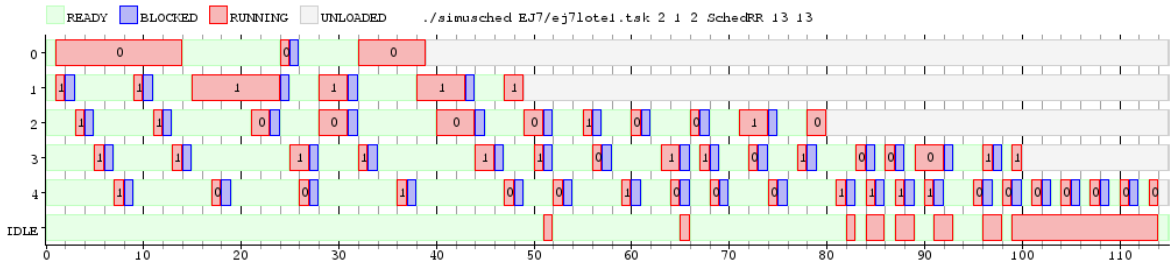
*Lote*1 - Turnaround - 2 core - Quantum igual a 9

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 se deben a que este tipo de tarea presenta una pseudoaleatoriedad notoria.



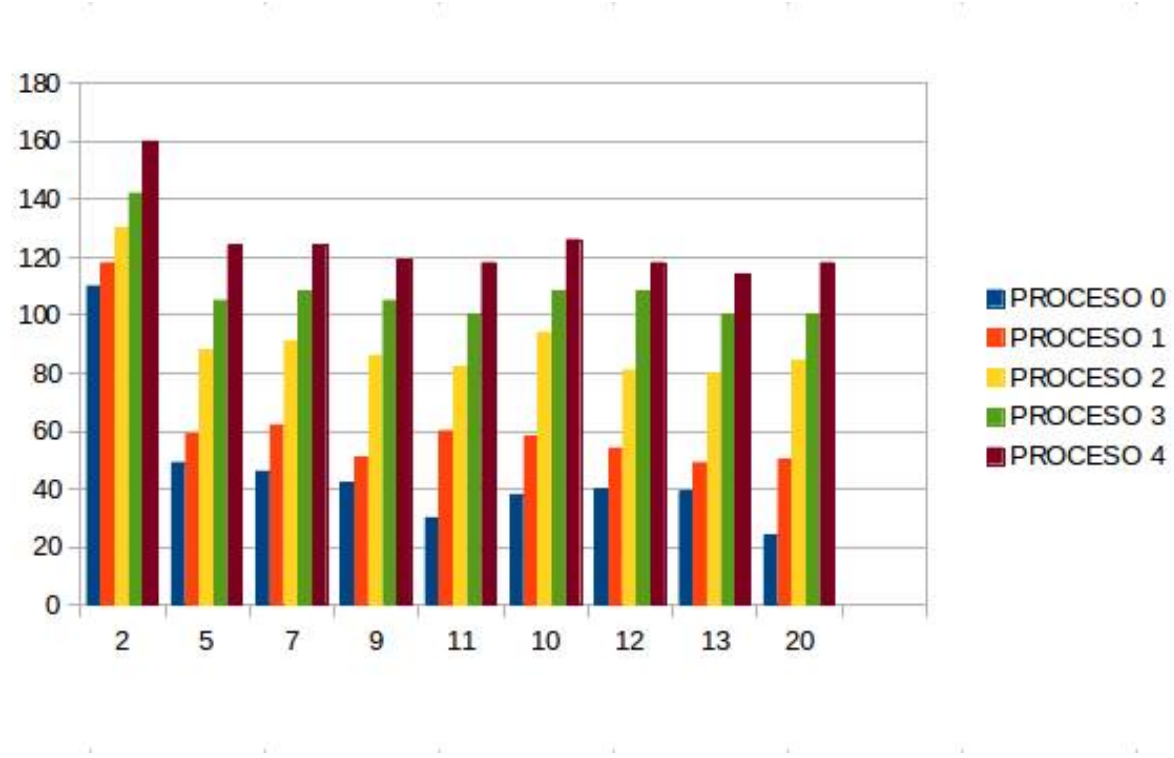
*Lote*1 - Turnaround - 2 core - Quantum igual a 11



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

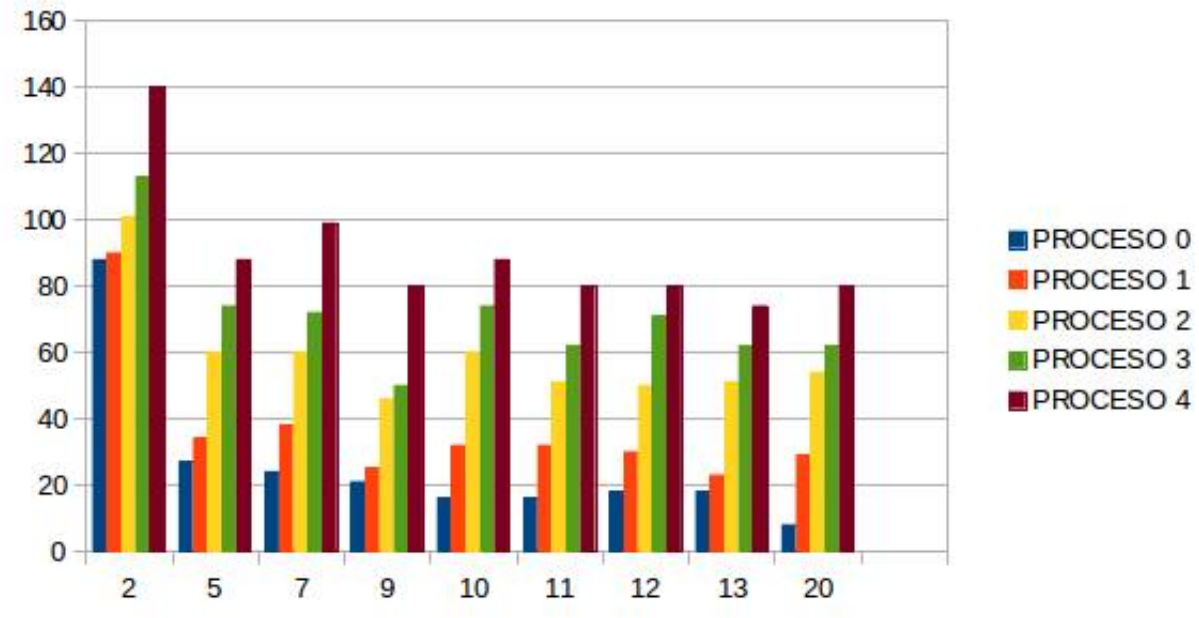
Como mencionamos, las mediciones tienden a estabilizarse con un *quantum* igual a 9, pero la mejor performance obtenida es con un *quantum* igual a 13, teniendo en cuenta como mencionamos la pseudoaleatoridad de este tipo de tarea.

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

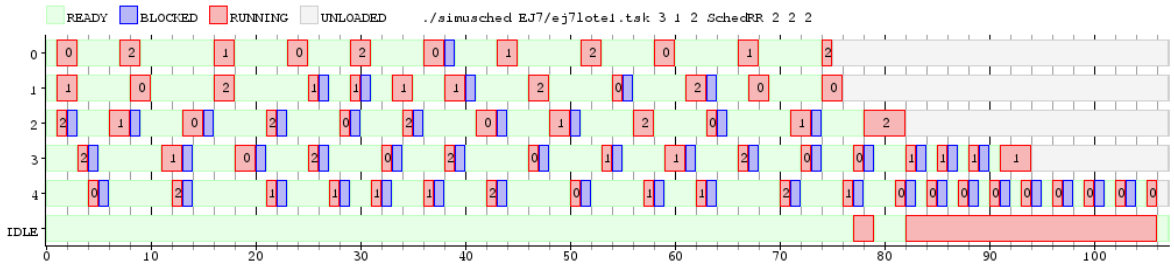


Waiting Time - 2 core
$EjeX = Quantum$
$EjeY = Tiempo$

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

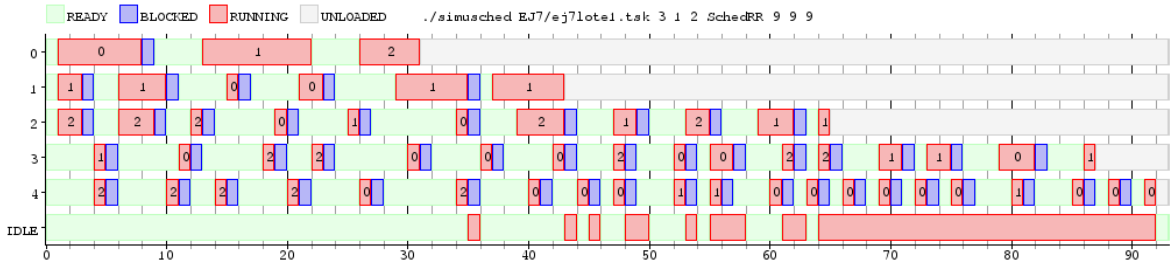
3 Core

A continuación, al igual que con 2 cores, mostraremos los resultados más relevantes:



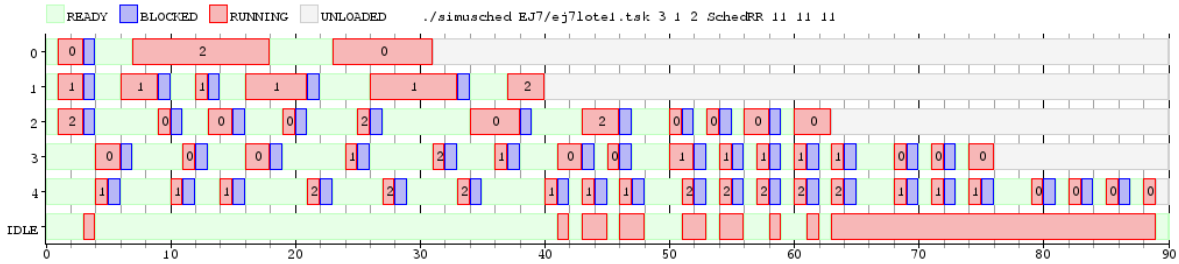
*Lote*1 - Turnaround - 3 core - Quantum igual a 2

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

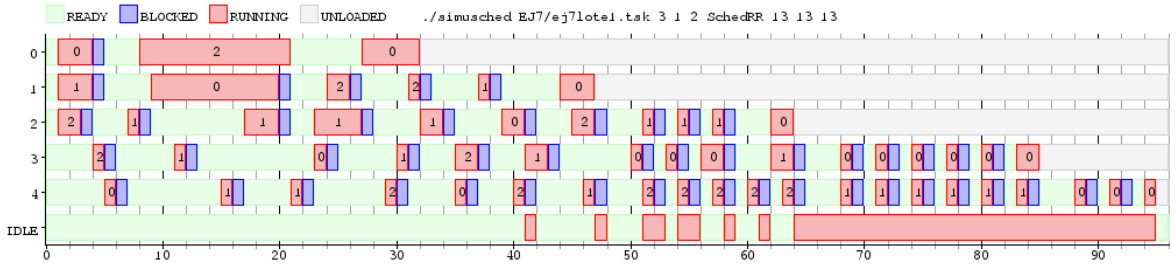


*Lote*1 - Turnaround - 3 core - Quantum igual a 9

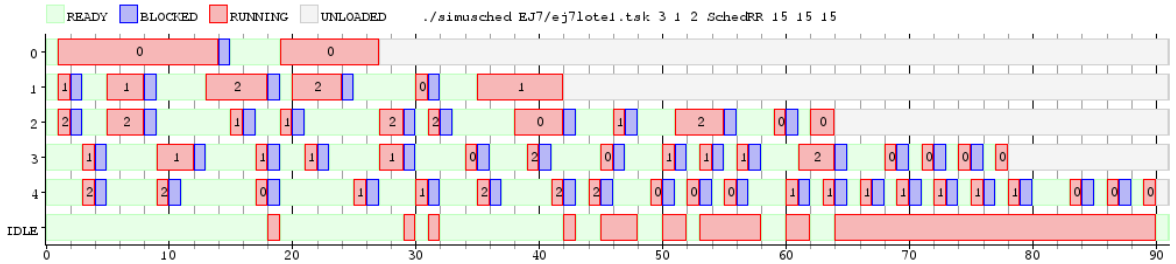
La performance empieza a mejorar a medida que el *quantum* aumenta.



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

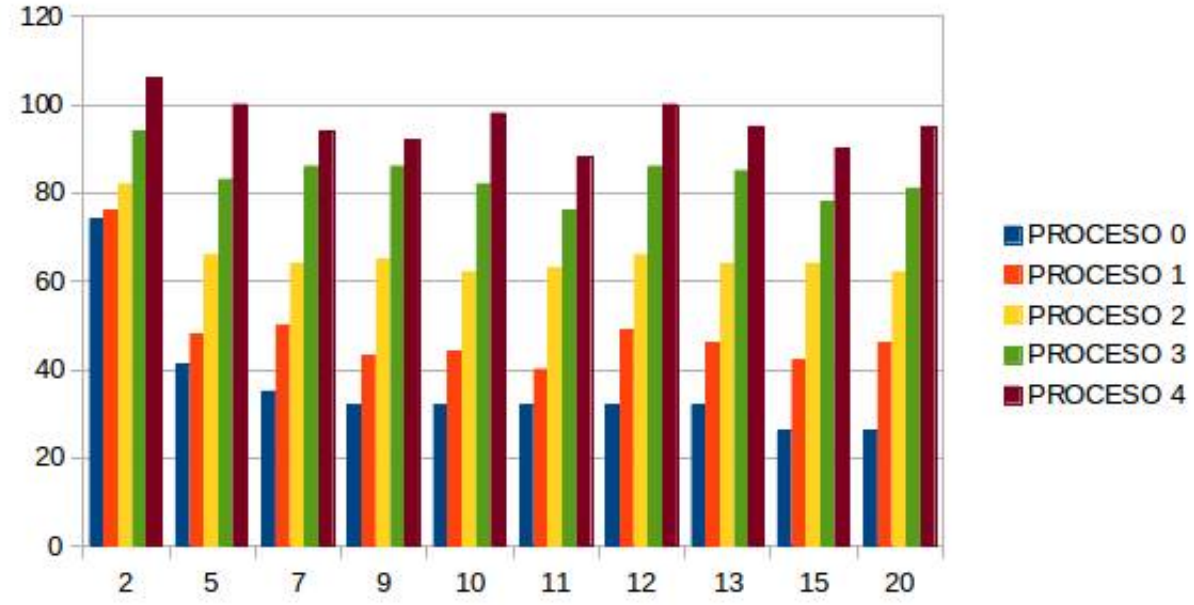


*Lote*1 - Turnaround - 3 core - Quantum igual a 13



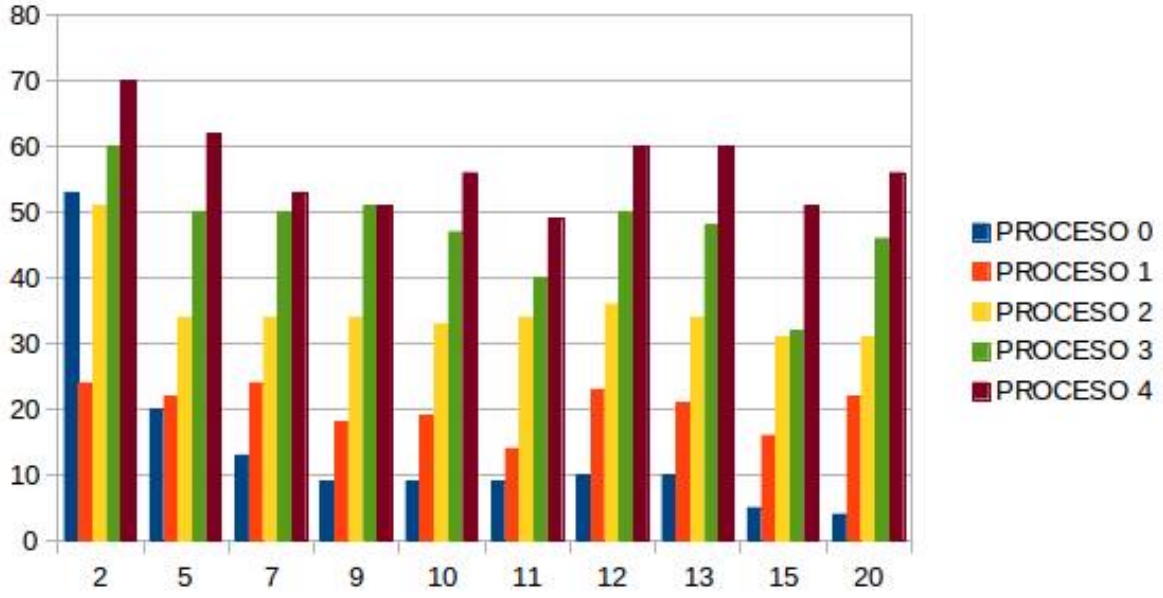
*Lote*1 - Turnaround - 3 core - Quantum igual a 15

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 el tipo de tarea con la que se estuvo trabajando, 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 métrica,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 presumir 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 puntual, aumentar el valor del quantum también mejora la performance, igualmente, como vimos, a partir de cierto valor de quantum, las mejoras en la performance tienden a estabilizarse y dejan de ser muy significativas. Esto se produce a que las tareas con mas cantidad de bloqueos en algun momento dejan de consumir todo el quantum otorgado si este es aumentado en su valor.

5.2.3 Ejercicio 8

La idea principal de esta nueva versión de $Round - Robin$ se centraliza en que no permita migración entre cores, esto se basa principalmente en utilizar una cola para cada núcleo por separado, y en cada cola respectiva se encolaran las tareas que fueron asignadas inicialmente a cada nucleo.
Para desarrollar este tipo de algoritmo, el cual denominaremos $RR2$, utilizamos estructuras puntuales, enunciadas a continuación:

- Un vector *quantum* y otro *quantumActual*, los cuales siguen cumpliendo la misma funcion que en Round-Robin 1.
- Un vector de colas denominado *colas*, en el cual, en la posición i encontraremos la cola correspondiente a ese núcleo de procesamiento.
- Un diccionario de *Bloqueados*, donde la clave contendra el número de core, y en definición la tareas bloqueadas de ese core. Esto nos beneficiara cuando haya que reubicarla en la cola de procesos ready.
- Un vector de enteros *cantidad*, que como la palabra lo define, tendrá en cada posición i la totalidad de las tareas, ya sea bloqueadas, activas o en estado ready que tiene asignado ese core, beneficiandonos la determinación del núcleo al que se le asignará la tarea al momento de cargarla.

Cuando se carga una tarea, previamente, se chequeará que core tiene menor cantidad de procesos totales asignados (aqui es donde el vector *cantidad* entra en juego). Una vez que se obtiene este nucleo, se agrega la tarea a la cola correspondiente y se actualiza la cantidad sumando una unidad.

Al bloquearse un proceso, se define una nueva entrada en el diccionario *bloqueados* con el pid y el nucleo correspondiente. De esta forma, al desbloquearse, colocamos la tarea en la cola del core correspondiente y eliminamos la entrada del diccionario. Así logramos resolver el inconveniente de la nula migración entre nucleos.

Finalmente, cuando una tarea finaliza, la quitamos y descontamos una unidad a la posición i del vector *cantidad*. Esta es la única vez, en la cual se descuenta. Aunque una tarea se bloquee, la misma seguirá contando en el vector. De esta forma se cumplirá, que las tareas son asignadas a los cores con menor cantidad de tareas.
Luego de realizar dicha implementación, en comparación al Round-Robin original, hemos conjeturado las siguientes hipótesis:

1. Dados un mismo lote de tareas y una misma configuración del scheduler (mismos costo en cambio de contexto y quantum) un único núcleo de procesamiento, ambos algoritmos deben comportarse de la misma manera.
2. Comportamiento menos eficiente en el RR2 con respecto al paralelismo, ya que al no permitir migración de nucleos este se pierde.
3. Comportamiento más eficiente en el RR2 con lotes de tareas que se bloquean un gran numero de veces. Esto surge ya que el Round-Robin original, es mas proclive a realizar cambios de contexto con la posibilidad de darse un cambio de core.

Por consiguiente, procederemos a demostrar lo conjeturado.

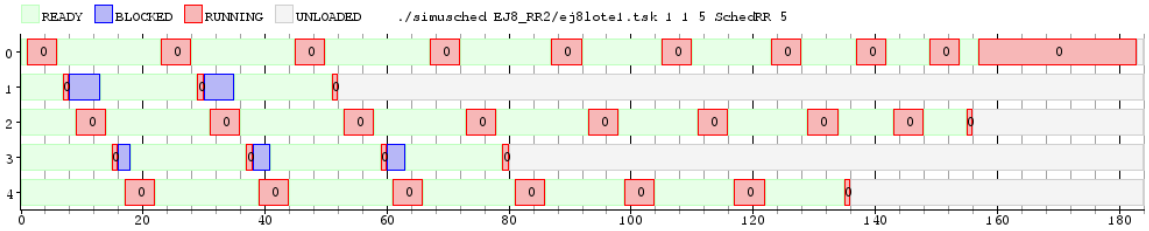
Iniciando con nuestra primer conjetura:

Dados un mismo lote de tareas y una misma configuración del scheduler (mismos costo en cambio de contexto y quantum) un único núcleo de procesamiento, ambos algoritmos deben comportarse de la misma manera.

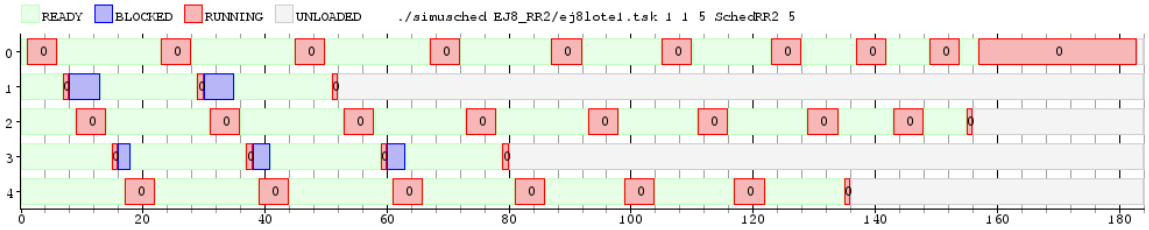
Trabajando con el lote que mencionamos a continuación:

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

Utilizando un *quantum* igual a 5 y un cambio de contexto igual a 1 obtuvimos resultados muy marcados, viendose notoriamente lo que queremos demostrar.



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

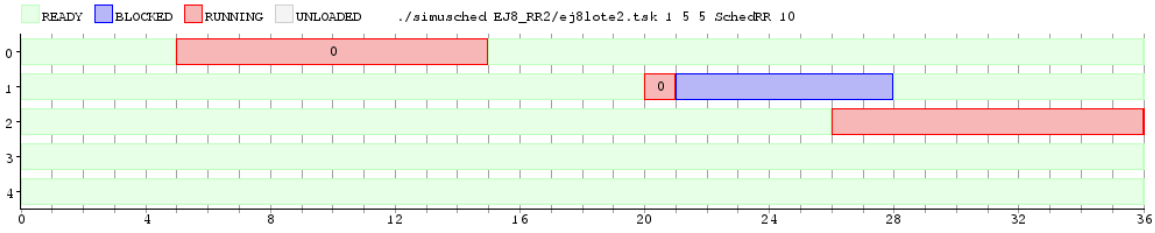


*Lote*1 - Round Robin 2 - 1 core - Quantum = 5 - cambio de contexto = 1

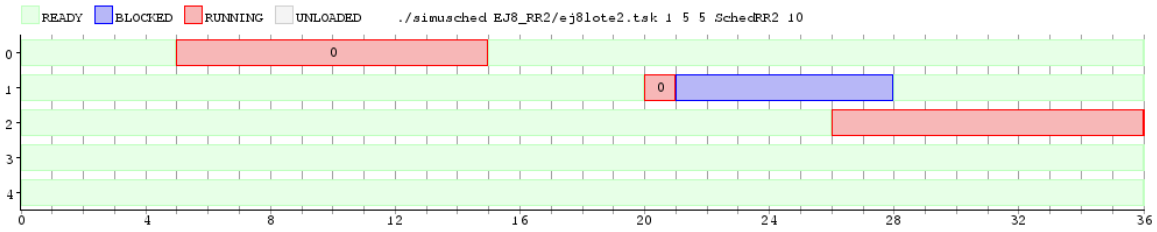
Continuando con un lote distinto para ser mas precisos:

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

Llegamos a lo siguiente:



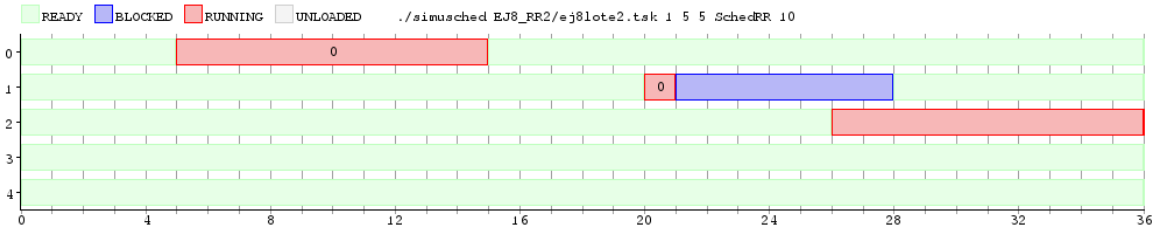
*Lote*2 - Round Robin - 1 core - Quantum = 10 - cambio de contexto = 5



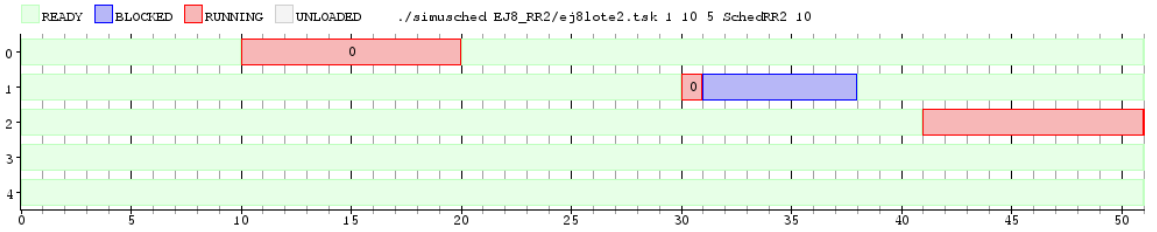
*Lote*2 - Round Robin 2 - 1 core - Quantum = 10 - cambio de contexto = 5

Viendose nuevamente, la igualdad que mencionamos. Hemos podido ver a su vez, que la única forma en la que los resultados sean distintos con el mismo lote seria modificando o la cantidad de *quantum* o la unidad de cambio de contexto entre uno y otro.

Aqui, un ejémplo para mayor precisión:



*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

Concluimos entonces que, al trabajar con colas globales o no, utilizando un único núcleo y mismo lote quantum y cambio de contexto, ambos schedulers se comportan de la misma manera.

Procedemos a demostrar la segunda conjetura:

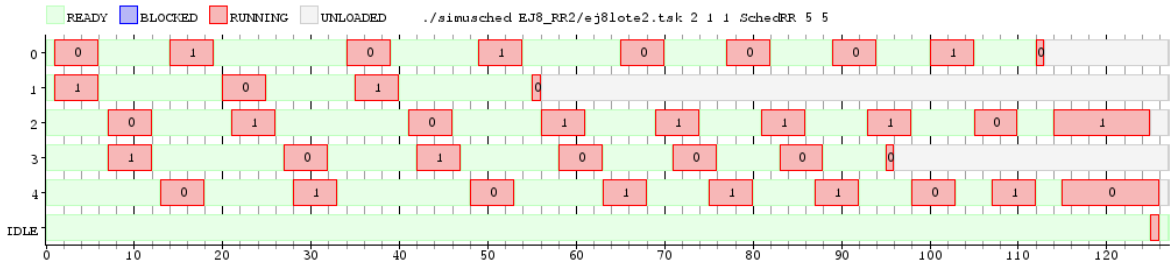
Comportamiento menos eficiente en el RR2 con respecto al paralelismo, ya que al no permitir migración de núcleos este lo pierde

Para demostrar esta conjetura trabajamos con procesos que demanden mas uso del cpu como lo son las *taskCPU*

Un ejemplo de los lotes utilizados fue el siguiente:

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

Obteniendo los siguientes datos relevantes:



*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

Se puede ver en estos diagramas como la implementación del Round Robin original trabaja mejor finalizando la ejecución de las tareas hasta 50 milisegundos antes.
Esto se da por la falta de paralelismo de el RR2 ya que al ser asignados los procesos a cada core, cuando uno de los dos finaliza, este queda ocioso ya que no existe la posibilidad de migrar procesos.

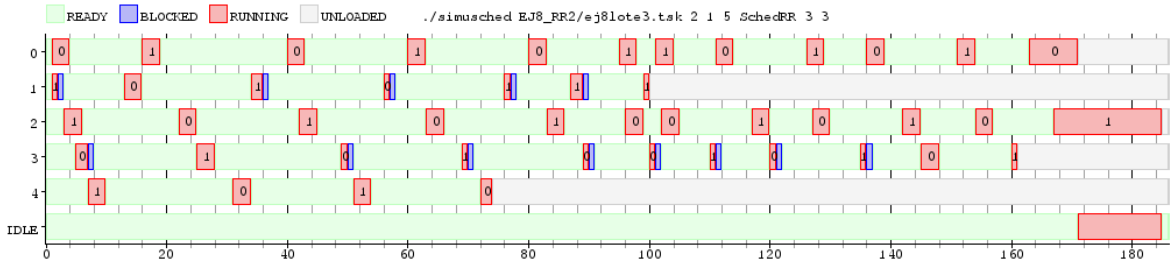
Por ultimo, nuestra tercer y ultima conjetura:

Comportamiento mas eficiente en el RR2 con lotes de tareas que se bloquean un gran numero de veces

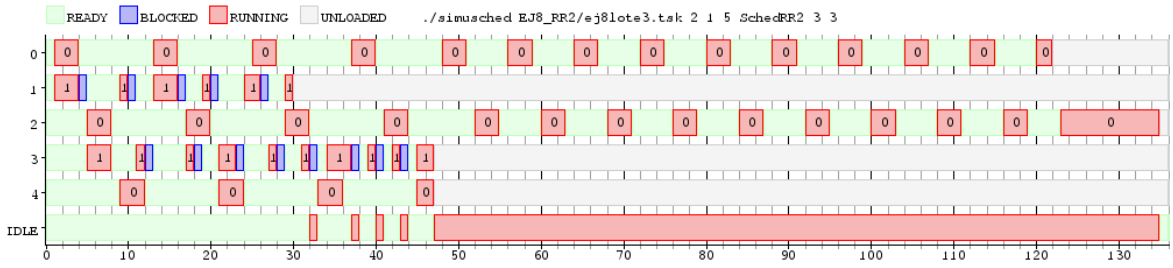
Para esta conjetura, trabajamos con lotes de tareas que utilicen el CPU y se bloqueen muchas veces para poder demostrar la mejor performance del RR2.

A continuación un ejemplo, con el siguiente lote:

```
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 puede observar por los diagramas como el RR2 tiene una mejor performance en este estilo de lotes llegando a finalizar las ejecuciones hasta 50 milisegundos antes que el Round-Robin original.
Como el Round-Robin original tiene pérdida de tiempo con el cambio de contexto y migración de tareas este empeora su performance en comparación al RR2 que no admite este tipo de migración es notorio la superioridad en relación a nuestra conjetura.

Podemos concluir luego de estas demostraciones que, el Round-Robin original es ampliamente superior desde el punto de vista de la performance que se obtiene al trabajar con tareas que demanden mucho uso del CPU, mientras que el RR2 es ampliamente mejor cuando se utilicen tareas que se bloqueen por un tiempo considerable.

5.2.4 Ejercicio 9

Luego del estudio sobre la relación de ambos algoritmos pudimos conjeturar las siguientes hipotesis:

- El Scheduler SchedFixed da prioridad a las tareas que menor período tengan.
- El Scheduler SchedDynamic da prioridad a las tareas que esten próximas a terminar su deadline.
- El Scheduler SchedFixed al dar prioridad a tareas con menor período, puede producir inanición sin importar las tareas que lleguen con un deadline alto y necesidad de correr de inmediato.

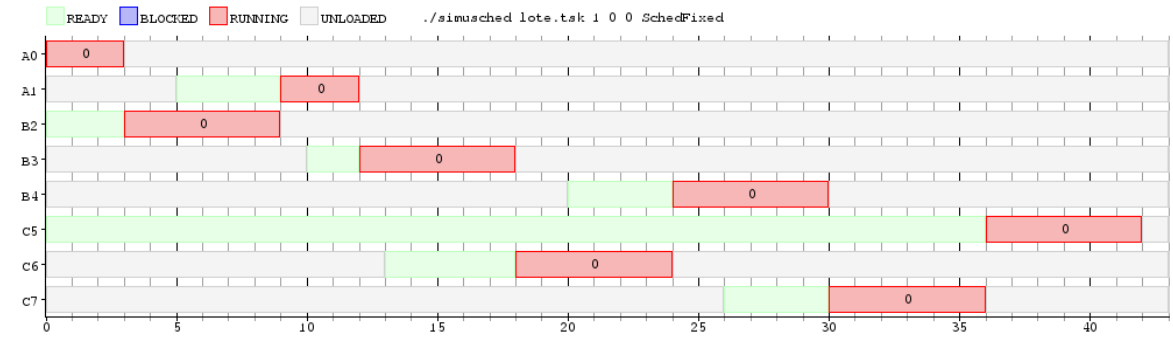
Iniciando con nuestra primer conjetura:

El Scheduler SchedFixed da prioridad a las tareas que menor período tengan.

Trabajando con el lote:

```
&A2,5,2
&B3,10,5
&C3,13,5
```

LLegamos a los siguientes resultados:



*Lote*1 - SchedFixed - 1 core

Se puede observar como las tareas de familia A que presentan un menor período en comparación son ejecutadas mas rapidamente.

Continuando con nuestra segunda conjetura:

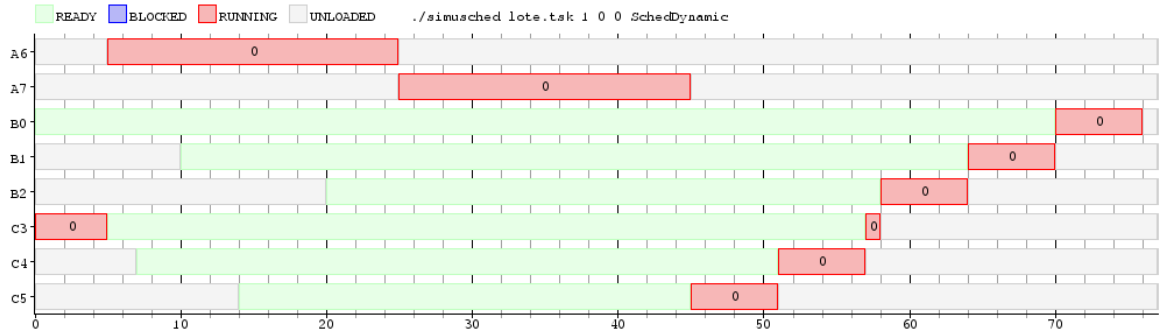
El Scheduler SchedDynamic da prioridad a las tareas que esten próximas a terminar su deadline

Para demostrar esta conjetura, trabajamos tanto con periodos como con tiempo de ejecucion mas largos.

Un ejemplo de uno de los lotes que utilizamos fue:

```
&B3,10,5
&C3,7,5
@5:
&A2,20,19
```

Con lo que, recibimos ciertos datos relevantes:



*Lote*2 - SchedDynamic - 1 core

Aquí, podemos observar como tareas con período más largo en comparación pero con mayor tiempo de ejecución a su vez, tienen mayor prioridad y hasta posibilidad de desalojar otra tarea para ser ejecutada de inmediato. Se puede ver en este ejemplo como la tarea A que ingresa en un tiempo @5 con un período alto pero también un run time alto hace que el scheduler, desaloje la que esta corriendo y ejecute esta familia de tareas.

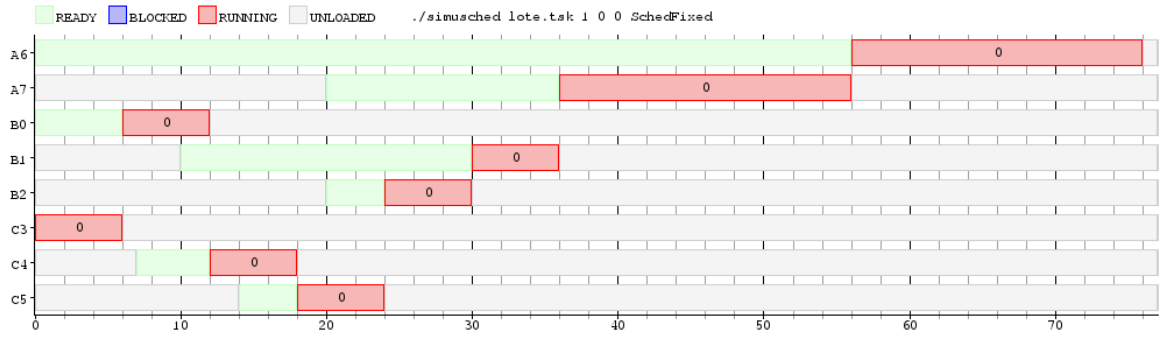
Con estas dos hipótesis demostradas, nos llevo a la conclusión de nuestra 3º, en la cual como mencionamos,**el Scheduler SchedFixed al dar prioridad a tareas con período más corto, puede dejar de lado tareas con necesidad de ejecución inmediata por tener un deadline próximo a terminar, mientras que, el Scheduler SchedDynamic optimiza esta situación dando prioridad a las tareas que esten próximas a finalizar su deadline**

Un ejemplo de lote de lo conjeturado sería el siguiente:

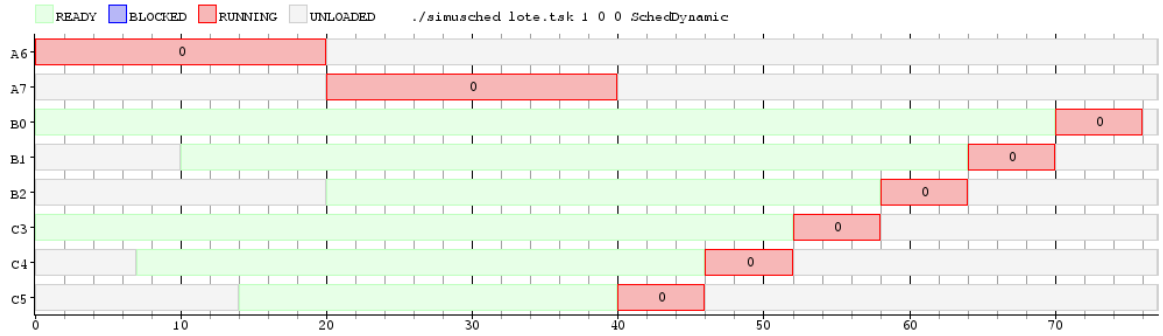
```
&B3,10,5
&C3,7,5
&A2,20,19
```

Como vemos, tenemos tareas con período relativamente corto, y otras con período más largo, pero con su tiempo de ejecución más amplio también.

Con este tipo de lote llegamos a los siguientes resultados:



*Lote*3 - SchedFixed - 1 core



*Lote*3 - SchedDynamic - 1 core

Se puede observar lo que hemos enunciado y la negatividad que presenta el SchedFixed en comparación al realizar ejecuciones con tareas con periodos cortos y otras con períodos mas amplios pero con tiempos de ejecución similares al valor de su período. Dandole el privilegio de ejecución a las más cortas que a las otras. En este ejemplo puntualmente, el SchedFixed deja a la tarea de familia A perder hasta 3 veces su período a pesar de estar próxima a finalizar su deadline, dando prioridad a las de menor período.

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