



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I - Scheduling

16 / 9 / 2014

Sistemas Operativos

Integrante	LU	Correo electrónico
Straminsky, Axel	769/11	axelstraminsky@gmail.com
Chapresto, Matias		matiaschapresto@gmail.com
Torres, Sebastian	723/06	sebatorres1987@hotmail.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. Parte I: Entendiendo el simulador *simusched*

1.1. Introducción

El objetivo de esta parte es familiarizarse con el simulador *simusched*, el cual sirve para ver el comportamiento de distintos lotes de procesos bajo distintas políticas de scheduling. Adicionalmente se puede especificar la cantidad de cores a disposición de los procesos, y los costos de ciertas acciones como hacer un cambio de contexto, o cambiar un proceso para que se ejecute en otro core.

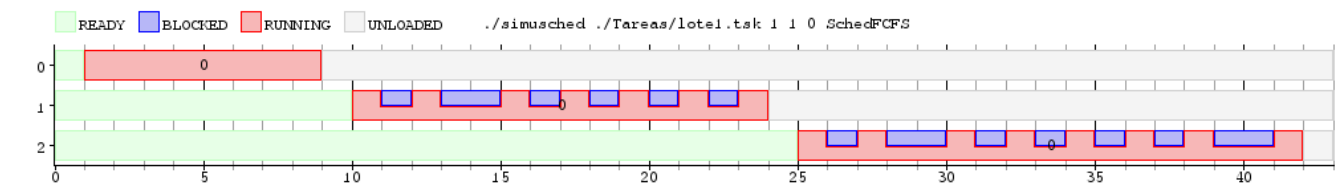
1.2. Ejercicio 1

El objetivo de este ejercicio es implementar una tarea de tipo **TaskConsole**, la cual debe simular ser una tarea interactiva. Para esto, la tarea realiza n llamadas bloqueantes, cada una con una duración al azar entre $bmin$ y $bmax$, ambos especificados por parámetro. La implementación de esta función es bastante directa, y básicamente consiste en inicializar el generador de números aleatorios con el parámetro *time(NULL)*, es decir, con la fecha actual al momento de ejecutarse la función. Luego, se realizan n llamadas bloqueantes con una duración al azar entre $bmin$ y $bmax$, mediante la fórmula $(rand() \% (bmax - bmin)) + bmin$, utilizando la función *uso_IO*. Para más detalles, consultar la implementación en el archivo *tasks.cpp*.

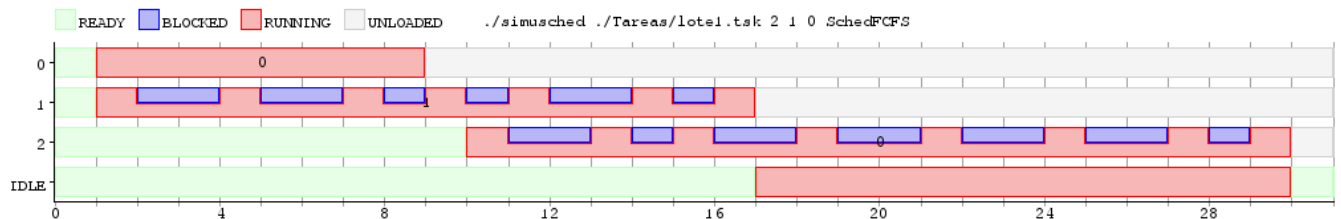
1.3. Ejercicio 2

El objetivo de este ejercicio es ejecutar un lote de tareas, una intensiva en CPU y las otras 2 de tipo interactivo (**TaskConsole**), con la política de scheduling **FCFS**, y observar y graficar los resultados, variando la cantidad de cores, con un costo de cambio de contexto igual a 1, y costo de migración igual a 0.

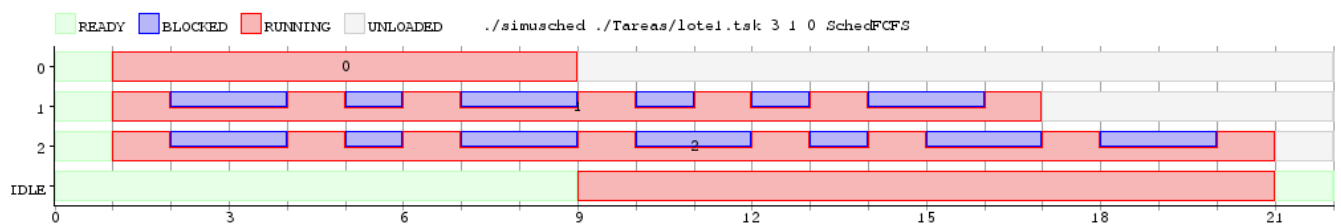
El lote de tarea que utilizamos es el *lote1.tsk*. A continuación se pueden ver los gráficos:



1 core.



2 cores.



3 core.

2. Parte II: Extendiendo el simulador con nuevos schedulers

2.1. Introducción

En esta sección se extiende el simulador con un nuevo algoritmo de scheduling, *Round Robin*, y se lo testea con diversos lotes de tareas.

2.2. Ejercicio 3

El objetivo de este ejercicio es implementar la política de scheduling *Round Robin*. La función más importante es `tick(cpu, motivo)`, cuya implementación se describe a continuación: si el motivo es **TICK** o **BLOCK**, se aumenta el contador de ticks del core correspondiente. Si este contador supera el quantum del core, se vuelve a poner el contador en 0, se encola la tarea actual y comienza a ejecutarse la siguiente tarea en la cola; caso contrario, se sigue ejecutando la tarea actual.

Si el motivo es **EXIT**, sencillamente se devuelve la próxima tarea en la cola, sin encolar nuevamente la tarea actual. En caso de no haber más tareas, se ejecuta **IDLE_TASK**.

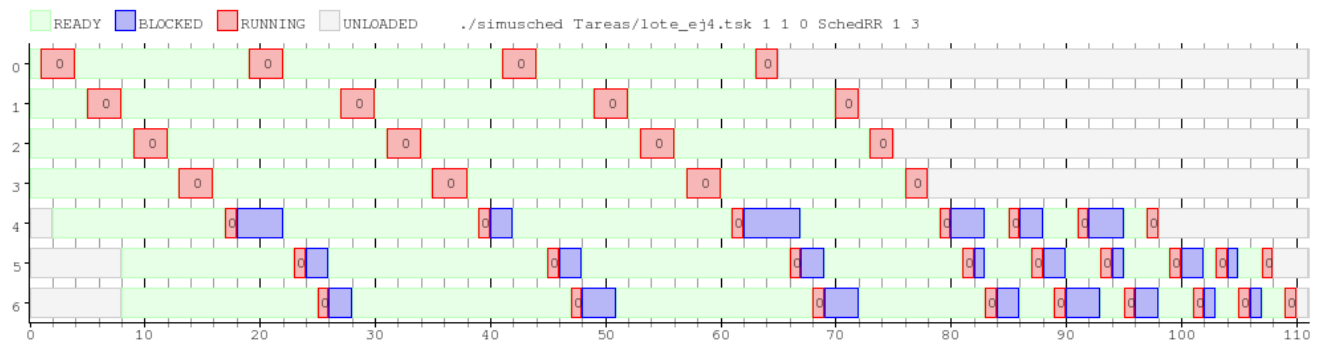
Para más detalles, consultar la implementación en el archivo `sched_rr.cpp`.

2.3. Ejercicio 4

En esta parte nos vamos a focalizar en la utilización de la política de scheduling implementada anteriormente para mostrar cómo se comporta la misma con diversos quantums y cantidad de cores. La intención es mostrar cuán eficiente o ineficiente puede ser una misma política tan solo con variar el quantum y mostrar que la elección del mismo es muy importante.

Para tal motivo vamos a usar el lote `lote_ej4.tsk` que contiene 4 tareas intensas en CPU y 3 interactivas. Las de CPU llegan en el momento 0, luego una interactiva en el momento 2 y otras dos más en el momento 8.

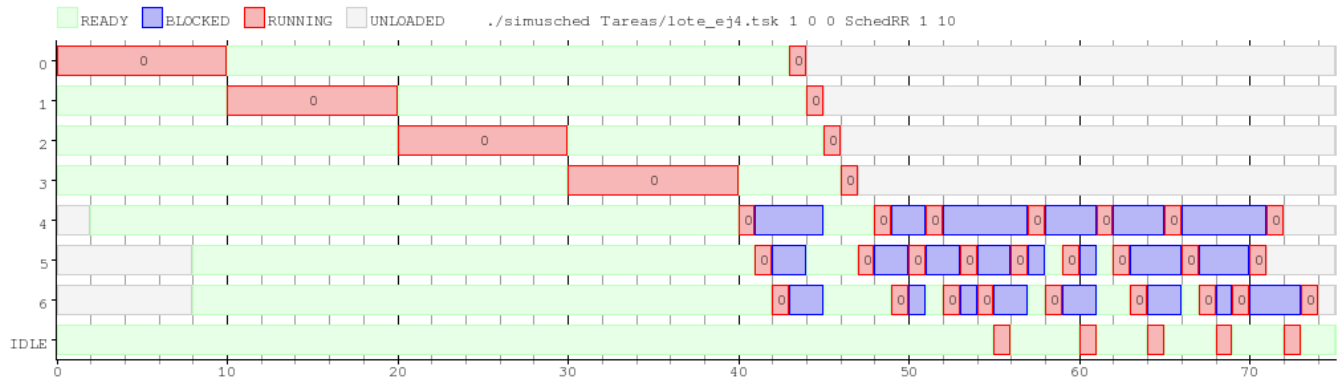
Como primera política vamos a tomar un RR con quantum 3, costo de cambio de contexto de 1 y un sólo un core. El procesamiento de dicho lote arroja el siguiente gráfico:



$$\text{Cores} = 1, \text{Quantum} = 3, \text{CS} = 1.$$

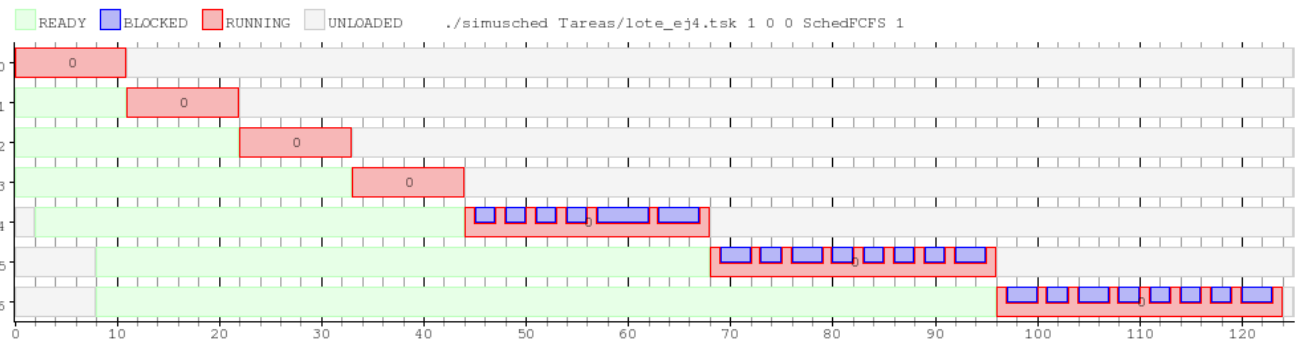
Como podemos apreciar, se ejecutan en una primera pasada las tareas de cpu (de la 0 a la 3 inclusive) que no tienen bloqueos. Luego se ejecuta la tarea 4, que llegó en el momento 2. Luego en lugar de ejecutarse la 5 o la 6 (que llegaron en el momento 8) vuelve a ejecutarse la tarea 0, luego la 1 y recién luego de esta se ejecuta la número 5. Esto ocurre porque cuando termina de ejecutarse por primera vez la tarea 0 vuelve a ser encolada porque su quantum terminó y todavía queda más por procesar. Cuando esto ocurre, todavía las tareas 5 y 6 no llegaron a la cola. Esto sucede también con la tarea 1. A mitad de la primera ejecución de la tarea 2 es cuando llegan las tareas 5 y 6 y por eso en la segunda corrida (momento 13) las tareas 5 y 6 se ejecutan después de las 0 y 1. Conforme pasan los ticks vemos como el gráfico va siguiendo un cierto patrón de tipo escalera, característico de Round Robin, ya que usa un esquema de despacho de tareas circular.

El siguiente experimento que queremos mostrar es qué sucede cuando el quantum de un esquema de RR es muy grande. A priori un quantum muy grande no sería muy conveniente ya que se estaría aproximando a un esquema de FCFS (en el caso de tareas intensas en CPU). A continuación vemos un experimento con un quantum de 10.



$Cores = 1, Quantum = 10, CS = 0.$

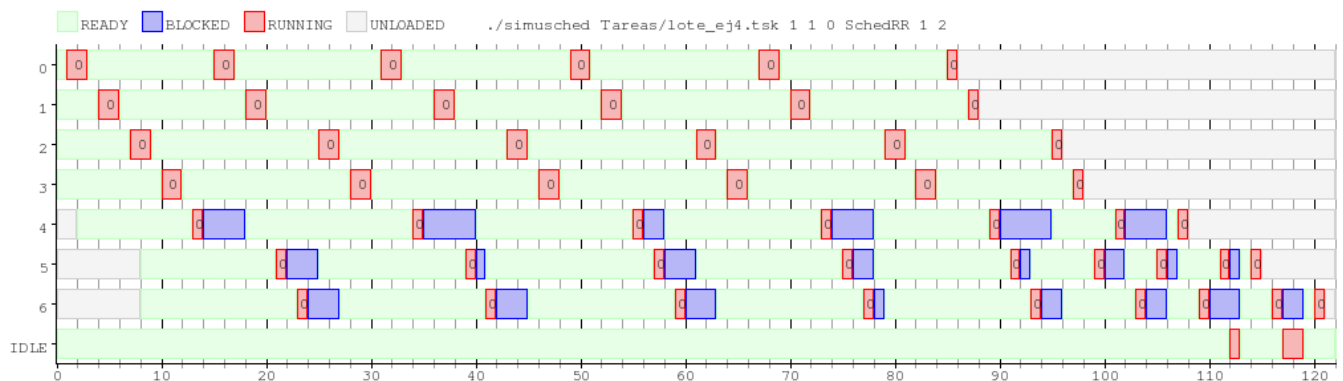
Y a continuación el mismo lote de tareas procesado con FCFS.



$Cores = 1, Quantum = 10, CS = 0.$

Como se puede apreciar, un esquema de RR con un quantum alto tiende a comportarse de una manera similar al FCFS con la importante salvedad de que RR es *starvation-free*, debido a que el quantum puede ser muy grande pero es finito.

Por otro lado, un quantum muy pequeño tampoco es bueno. Esto se debe a que, si asumimos un costo de cambio de contexto mayor a 0, se va a desperdiciar mucho tiempo en cambiar de contexto y eso entorpecerá el rendimiento general del sistema.



$Cores = 1, Quantum = 2, CS = 1.$

Si comparamos este último gráfico con el primero de la sección vemos que todas las tareas tardan más en finalizar su ejecución y la cantidad de tiempo en cambios de contexto es mayor, lo cual se traduce en un menor rendimiento.

3. Parte III: Evaluando los algoritmos de scheduling

3.1. Introducción

En esta sección se evalúan las políticas de scheduling implementadas, utilizando diversas métricas especificadas más adelante.

3.2. Ejercicio 6

El objetivo de este ejercicio es programar un tipo de tarea **TaskBatch**, que durante *total_cpu* ciclos, realice *cant_bloqueos* llamadas bloqueantes, en momentos elegidos pseudoaleatoriamente. La implementación es bastante directa, con la semilla del generador de números pseudoaleatorios inicializada con la fecha del sistema al momento de ejecutar la función. Las llamadas bloqueantes se lanzan si *rand()* devuelve un número impar. Para asegurarnos de que se ejecuten las *cant_bloqueos* llamadas bloqueantes antes de que se terminen los *total_cpu* ciclos, creamos procesos cuya cantidad de llamadas bloqueantes fuese menor o igual a la mitad de los ciclos totales de CPU.

Para más detalles, consultar la implementación en *tasks.cpp*.

3.3. Ejercicio 7

En este ejercicio debemos elegir 2 métricas diferentes y testear un lote de tareas **TaskBatch**, todas ellas con igual uso de CPU pero con diversas cantidades de bloqueos. El lote de tareas utilizado es el *lote3.tsk*.

Las métricas que elegimos fueron:

- Turnaround
- Waiting Time

Definidas en [Sil1] como:

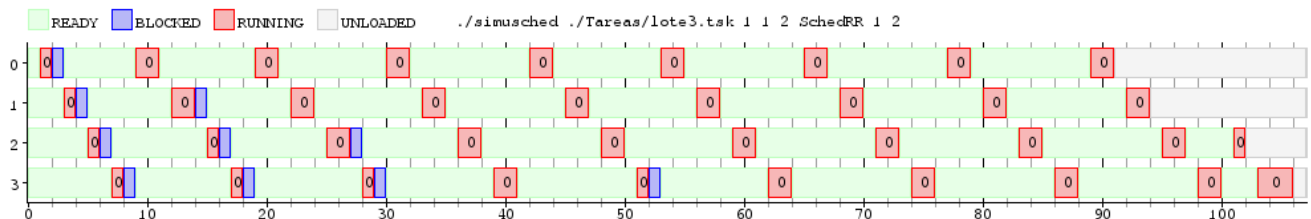
Turnaround: Es el intervalo de tiempo entre el momento en que el proceso comienza a ejecutarse por primera vez, hasta el momento en que el mismo termina. Es decir, es la suma de los períodos usados en esperar datos de memoria, en estar encolado en la “ready queue”, ejecutándose en la CPU, y haciendo E/S.

Waiting Time: Es el tiempo que un proceso se pasa encolado en la “ready queue”.

Elegimos estas métricas ya que, con el Turnaround, tenemos una visión global de cómo se comportan los procesos, mientras que con el Waiting Time podemos observar cómo un hecho más puntual, el tiempo que los procesos pasan encolados, impacta en el tiempo de ejecución total del proceso.

Para calcular el desvío standard utilizamos la fórmula de [WikSD]

A continuación se pueden observar los resultados de la experimentación:



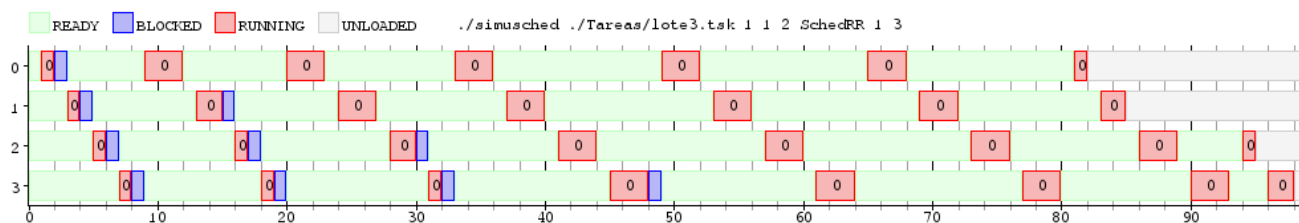
1 core, quantum = 2.

Turnaround:

P_0 : 90
 P_1 : 91
 P_2 : 97
 P_3 : 99
 Promedio: 94,25
 DS: 3,83

Waiting Time:

P_0 : 73
 P_1 : 75
 P_2 : 82
 P_3 : 85
 Promedio: 78,75
 DS: 4,92



1 core, quantum = 3.

Turnaround:

P_0 : 81

P_1 : 82

P_2 : 90

P_3 : 91

Promedio: 85,5

DS: 4,55

Waiting Time:

P_0 : 64

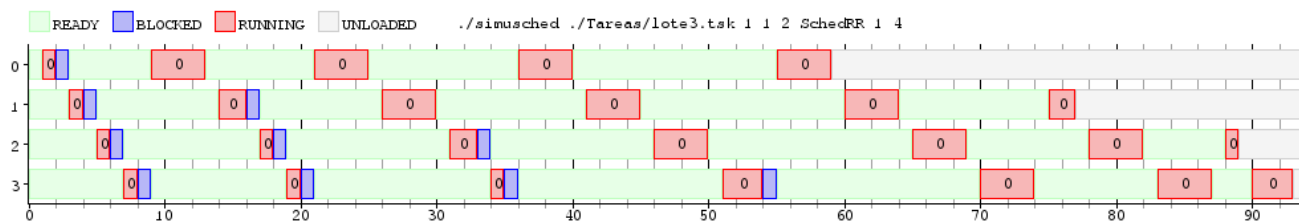
P_1 : 66

P_2 : 75

P_3 : 77

Promedio: 70,5

DS: 5,60



1 core, quantum = 4.

Turnaround:

P_0 : 58

P_1 : 74

P_2 : 84

P_3 : 86

Promedio: 75,5

DS: 10,33

Waiting Time:

P_0 : 41

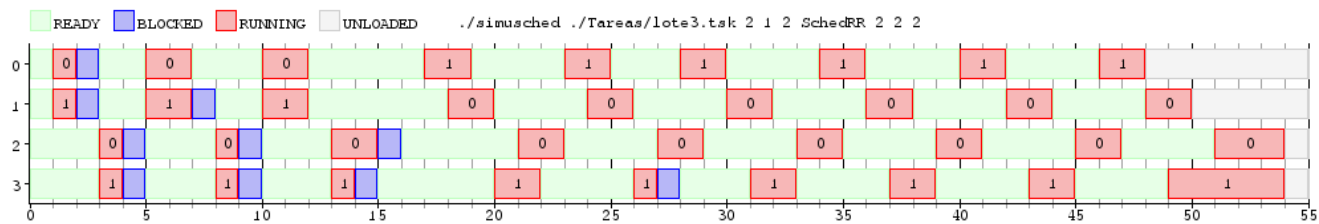
P_1 : 57

P_2 : 69

P_3 : 72

Promedio: 59,75

DS: 12,19



2 core, quantum = 2.

Turnaround:

P_0 : 47

P_1 : 49

P_2 : 51

P_3 : 51

Promedio: 49,5

DS: 1,66

Waiting Time:

P_0 : 30

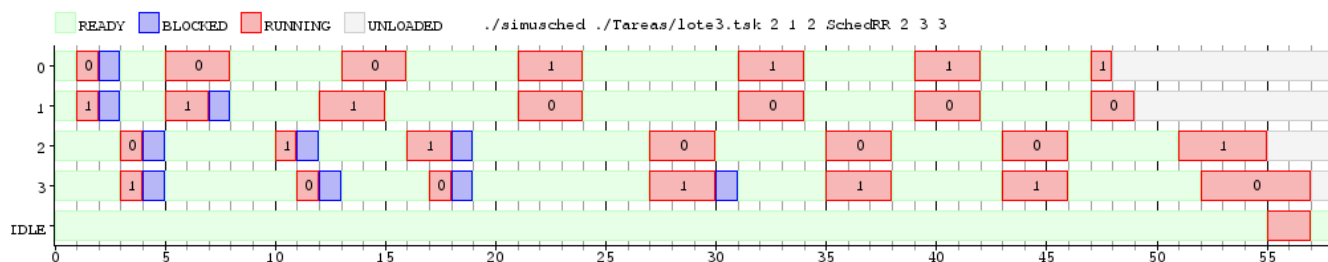
P_1 : 31

P_2 : 50

P_3 : 33

Promedio: 36

DS: 8,15



2 core, quantum = 3.

Turnaround:

P_0 : 47

P_1 : 48

P_2 : 52

P_3 : 54

Promedio: 50,25

DS: 2,86

Waiting Time:

P_0 : 30

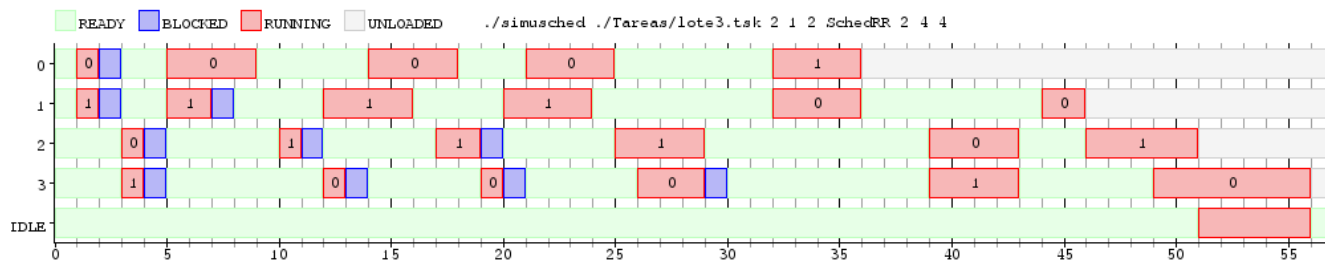
P_1 : 30

P_2 : 35

P_3 : 36

Promedio: 32,75

DS: 2,77



Turnaround:

 P_0 : 35 P_1 : 45 P_2 : 48 P_3 : 53

Promedio: 45,25

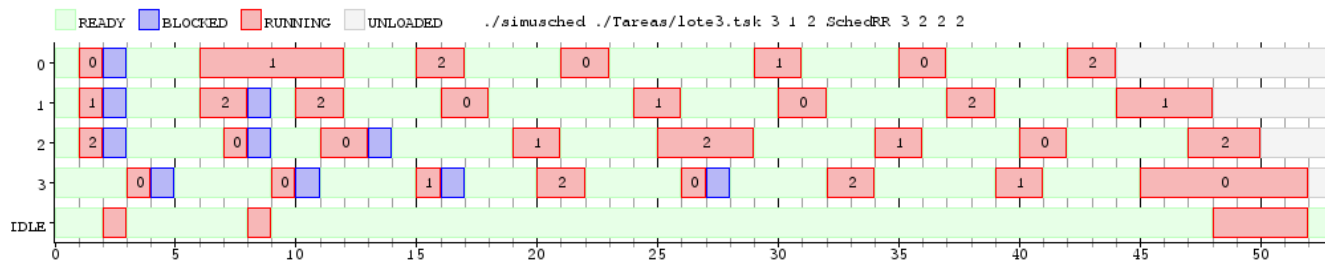
DS: 6,57

Waiting Time:

 P_0 : 18 P_1 : 27 P_2 : 31 P_3 : 35

Promedio: 27,75

DS: 6,30



Turnaround:

 P_0 : 43 P_1 : 47 P_2 : 49 P_3 : 49

Promedio: 47

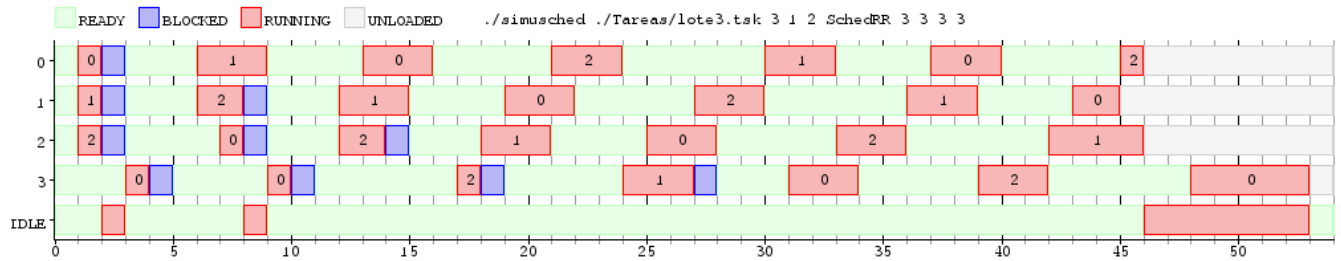
DS: 2,45

Waiting Time:

 P_0 : 26 P_1 : 29 P_2 : 30 P_3 : 31

Promedio: 29

DS: 1,87



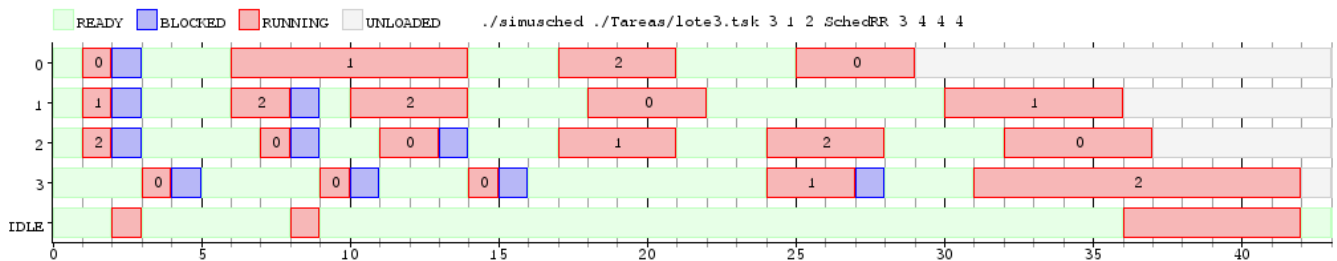
3 core, quantum = 3.

Turnaround:

P_0 : 45
 P_1 : 44
 P_2 : 45
 P_3 : 50
 Promedio: 46
 DS: 2,34

Waiting Time:

P_0 : 28
 P_1 : 26
 P_2 : 26
 P_3 : 32
 Promedio: 28
 DS: 2,45



3 core, quantum = 4.

Turnaround:

P_0 : 28
 P_1 : 35
 P_2 : 36
 P_3 : 39
 Promedio: 34,5
 DS: 4,03

Waiting Time:

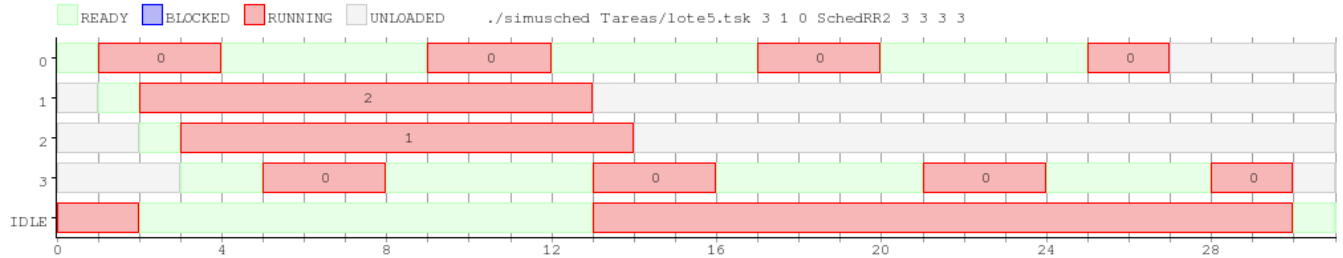
P_0 : 11
 P_1 : 17
 P_2 : 17
 P_3 : 21
 Promedio: 16,5
 DS: 3,57

3.4. Ejercicio 8

En la presente sección vamos a trabajar sobre un algoritmo de scheduling de tipo Round Robin pero que no permite la migración de procesos entre núcleos. Para tal motivo se utiliza una cola de **READY** para cada core, donde una vez que el proceso llega se moviliza sólo por esa cola.

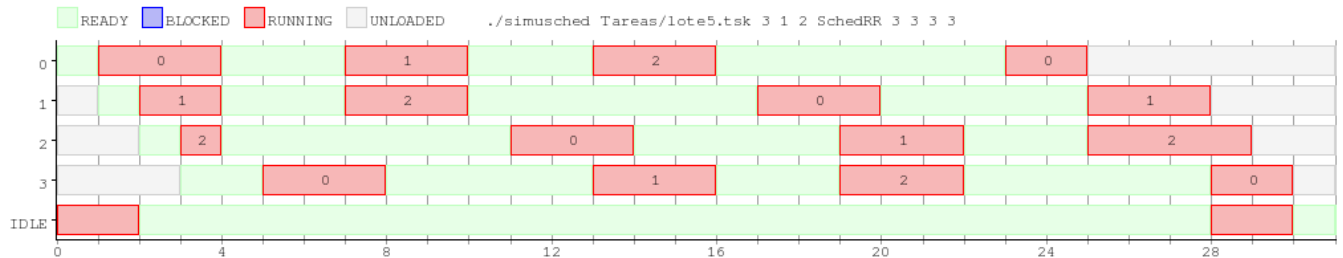
Cuando el proceso es bloqueado debemos tener una manera de saber a qué CPU pertenecía dicho proceso, y para tal fin cada core también tiene una cola de **BLOQUEADO**. Entonces, cuando un proceso es desbloqueado simplemente se lo busca en alguna de las listas **BLOQUEADO** de los cores, y una vez que se lo encuentra se lo agrega a la cola **READY** correspondiente al core que tenía esa tarea bloqueada. Esto hace que cada CPU tenga un esquema de Round Robin de un sólo core independiente del resto.

A continuación mostramos un gráfico correspondiente al procesamiento del lote de tareas *lote5.tsk* para este nuevo scheduler. Las tareas se encolan acorde a su momento de llegada y a la carga de los demás cores. Las tres primeras se cargan una en cada core, la cuarta se carga en el primer core ya que todos están igualmente cargados. Luego todas siguen ejecutándose en sus respectivos cores sin cambios de lugar como es esperado.



Cores = 3, Quantum = 3 cada core, CS = 1.

Ahora, con el objetivo de comparar con la política de scheduling de una sola cola, vamos a correr el mismo lote para ver cómo se comporta. A continuación se encuentra el gráfico con la distribución de las tareas.



Cores = 3, Quantum = 3 cada core, CS = 1, CI = 2.

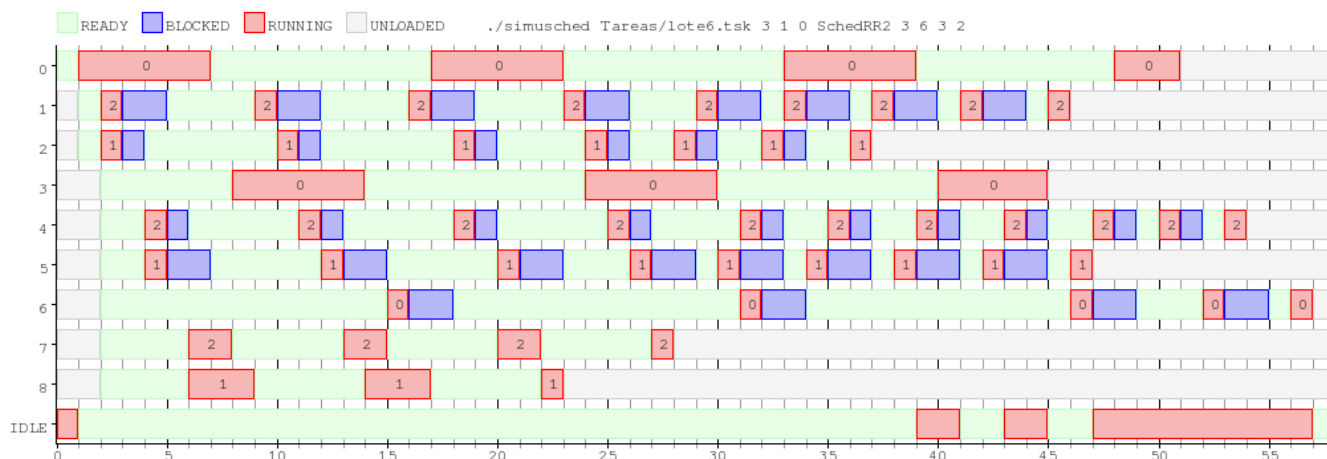
Un detalle que vale la pena mencionar es que las tareas desde que iniciar hasta que finalizar están asociadas a un solo core en el caso del gráfico uno. No ocurre así en el gráfico 2, en donde porciones de la tarea van cambiando de core, como se ve en el segundo gráfico.

Para este lote de tareas en particular podemos apreciar que **SchedRR2** y **SchedRR** finalizan la ejecución de todas las tareas en un tiempo similar. Pero en el primer caso dos de las tareas finalizan rápido, algo que no sucede en el segundo caso, sin contar que en **SchedRR2** no hay costos por cambio de core, que sí presenta **SchedRR**.

Por otro lado, si bien es cierto que **SchedRR2** termina dos de las tres tareas más rápido, también es cierto que esos cores quedan inactivos el resto del tiempo (para este lote) algo que no es deseable. Esta comparativa nos brinda un indicio de que si bien **SchedRR2** parece más eficiente, para algunos escenarios puede presentar desventajas con respecto a **SchedRR**, y dicho escenario es cuando un core se queda con muchas tareas pendientes y los demás no poseen ninguna. Para estos casos se podría implementar alguna política de balance de carga, es decir, si un core tiene muchas tareas y los demás ninguna, se podrían mover algunas tareas del core más ocupado para los de menos carga. En este caso estaríamos pagando la penalización del cambio de core (por el traslado de colas), pero es preferible eso antes que algún core quede inactivo.

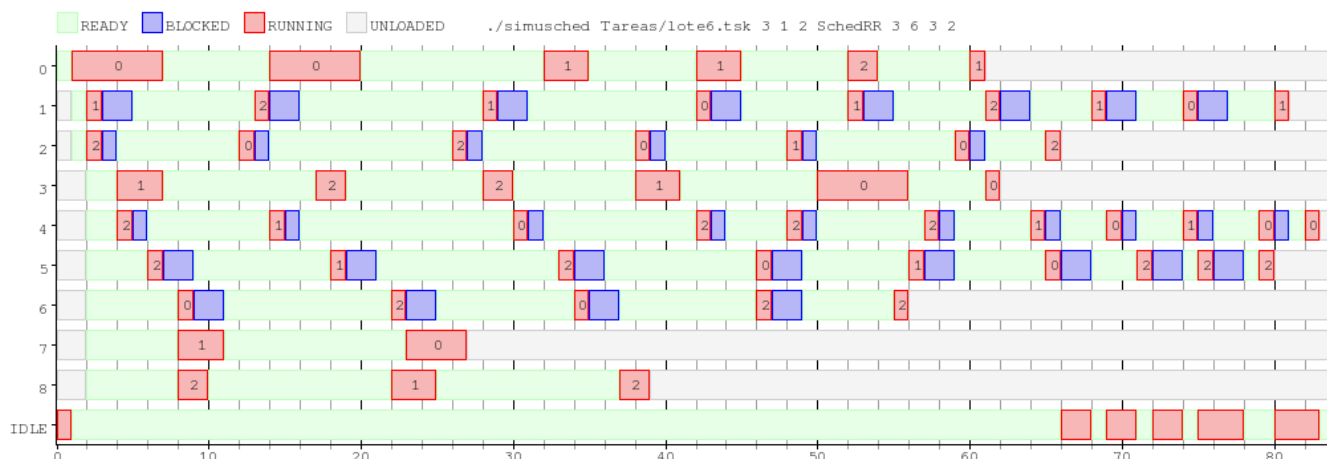
A continuación vamos a correr nuevamente los dos schedulers con el lote de tareas *lote6.tsk* que resultará en tareas pesadas para el core 0, y tareas más livianas para el resto. El primer core tendrá un quantum más grande que los otros. La idea es mostrar un escenario en donde puedo aprovechar cierta característica de los cores.

La idea detrás del diseño de este lote es construirlo para que el core 0 tome las tareas más pesadas (en cuanto a CPU) y el resto las más livianas. Además se configura la corrida del scheduler con tres cores, de los cuales el primero tiene un quantum más alto que los demás. Así queda el procesamiento del *lote6.tsk* con **SchedRR2**:



$Cores = 3$, $Quantum = 3$ cada core, $CS = 1$.

Y con **SchedRR** el procesamiento de dicho lote queda de la siguiente forma:



$Cores = 3$, $Quantum = 3$ cada core, $CS = 1$, $CI = 2$.

Vemos que para este lote, **SchedRR2** tiene mejor performance. Las tareas son finalizadas más rápido, y todo el lote es procesado más rápido que en **SchedRR**. Aún así en **SchedRR2** tenemos bastante tiempo de procesamiento en donde un procesador u otro permanece ocioso. Eso ocurre también en **SchedRR** pero es menor la cantidad de tiempo que los cores permanecen ociosos, aunque el desempeño final es peor.

3.5. Ejercicio 9

3.6. Ejercicio 10

4. Referencias

[Sil1] A. Silberschatz, *Operating System Concepts*, 4^o Ed., 1994, págs 135-136.

[WikSD] http://en.wikipedia.org/wiki/Standard_deviation#Basic_examples