DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Scheduling

16 de septiembre de 2014

Sistemas Operativos

Integrante	LU	Correo electrónico
Rodriguez, Pedro	197/12	pedro3110.jim@gmail.com
Benegas, Gonzalo Segundo	958/12	gsbenegas@gmail.com
Barrios, Leandro Ezequiel	404/11	ezequiel.barrios@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Esta página fue dejada en blanco intencionalmente.

Contents

1 Entendiendo el simulador simusched **1**

1.1 Ejercicio 1 1

1.2 Ejercicio 2 2

2 Extendiendo el simulador con nuevos schedulers **6**

2.3 Ejercicio 3 6

2.4 Ejercicio 4 7

2.5 Ejercicio 5 12

3 Evaluando los algoritmos de scheduling **13**

3.6 Ejercicio 6 13

3.7 Ejercicio 7 14

3.8 Ejercicio 8 15

3.9 Ejercicio 9 16

3.10 Ejercicio 10 18

Abstract

En el presente trabajo práctico estudiaremos algunos de los métodos más comúnmente utilizados en la actualidad por distintos sistemas operativos para manejar correcta y eficientemente los diversos procesos que se ejecutan concurrentemente en máquinas con uno o más procesadores.

Intentaremos detectar las ventajas y desventajas de cada método, así como los escenarios en los cuales uno puede ser más eficiente que otro. Para esto, dividimos el TP en tres partes: en la primera, presentamos el simulador `simusched` y lo corremos para algunas tareas específicas. En la segunda parte, extendemos el simulador con nuevos schedulers implementados por nosotros y finalmente, en la tercera parte evaluamos y analizamos los distintos algoritmos de scheduling ya presentados.

Esta página fue dejada en blanco intencionalmente.

Parte I Entendiendo el simulador simusched

1.1 Ejercicio 1

En este ejercicio, programamos la tarea TaskConsola, que simularía una tarea interactiva con el usuario.

Para la implementación de esta tarea, primero tuvimos que registrarla para que el simulador la reconozca en el archivo `tasks.cpp`, con la función `tasksinit(void)`. En ese mismo archivo implementamos la tarea, que para el proceso número *pid* recibe los tres parámetros: *n*, *bmin* y *bmax*. Simplemente hacemos un ciclo que cicle n veces y que cada vez tome un entero al azar[1] *r* entre *bmin* y *bmax*, y cada vez llamamos a la función `usoIO(pid, r)`, que simula hacer uso de dispositivos de entrada/salida.

Se denomina llamada bloqueante a aquellas llamadas en las que, si lo que esta solicita no está disponible, entonces el proceso llamador se queda bloqueado a la espera de un resultado. Es decir, no permite que otros procesos dependientes de éste sigan ejecutando. Por el contrario, en una llamada no bloqueante, si el proceso llamador no encuentra lo que estaba buscando, el proceso devuelve de todas formas algún resultado, permitiendo que otros procesos sigan ejecutando sin tener que esperar a que este proceso llamador reciba el resultado que está esperando.

[1] para elegir el número pseudo-aleatorio hacemos uso de la función `rand()` provista por la librerá standard de C.

1.2 Ejercicio 2

Para seguir entendiendo el funcionamiento del simulador simusched, ahora pasamos a escribir un lote de 3 tareas distintas: una intensiva en CPU y las otras dos de tipo interactivo. Vamos a ejecutar y graficar la simulación usando el algoritmo FCFS para 1, 2 y 3 núcleos.

En el caso de la tarea que hace uso intensivo del cpu, pudimos observar que usando dos cpu's, las tareas se logran ejecutar prácticamente en la mitad de tiempo (en el gráfico se ve claramente como se aprovecha el hecho de que hay dos cpu's, y en todo momento se puede observar cómo hay dos cpu's trabajando al mismo tiempo en dos tareas distintas. Cuando introdujimos un tercer cpu, sucedió exactamente lo mismo: cada vez que algún cpu terminaba de ejecutar un proceso, inmediatamente comenzaba a ejecutar el próximo proceso aún no atendido.

En el caso de las tareas interactivas, pudimos observar los momentos en los que se detectaba que algún procesador era bloqueado y cómo en ese tiempo el procesador se quedaba trabado en la misma tarea. Creemos que ese tiempo de espera podría ser aprovechado para seguir ejecutando algún otro proceso, implementando algún otro sistema de scheduling.

Otra cosa importante a destacar, es que en los tres lotes de tareas, cuando pasamos de usar un único core a usar dos, el tiempo total que se tardó en ejecutar todas las tareas disminuyó a la mitad. Mientras que cuando usamos tres cores, el tiempo no disminuyó a un tercio de lo que tardaba antes, que es lo que nosotros esperábamos. Esto nos da la pauta de que más procesadores no siempre mucha más eficiencia y velocidad: hay otros factores que son más importantes, como el algoritmo de scheduling utilizado, y la consideración de las características del lote de tareas que voy a querer correr.

A continuación, presentamos los diagramas de Gantt de los cuales hablamos:

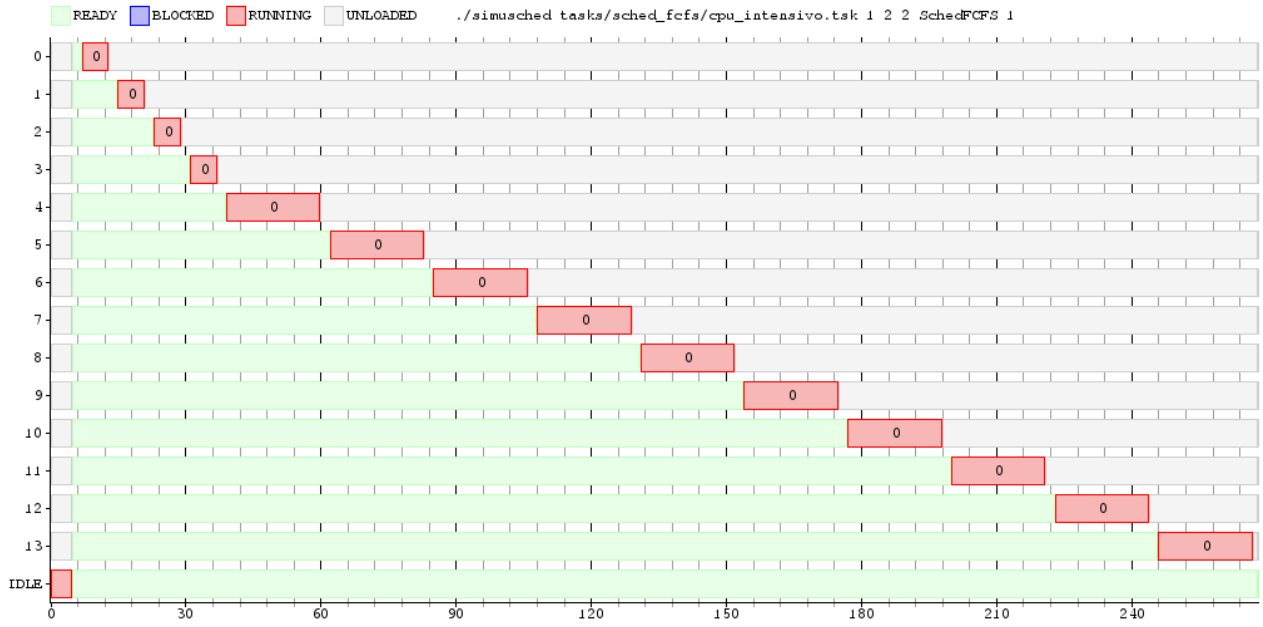

Figure 1.2.1: Intensivas en CPU

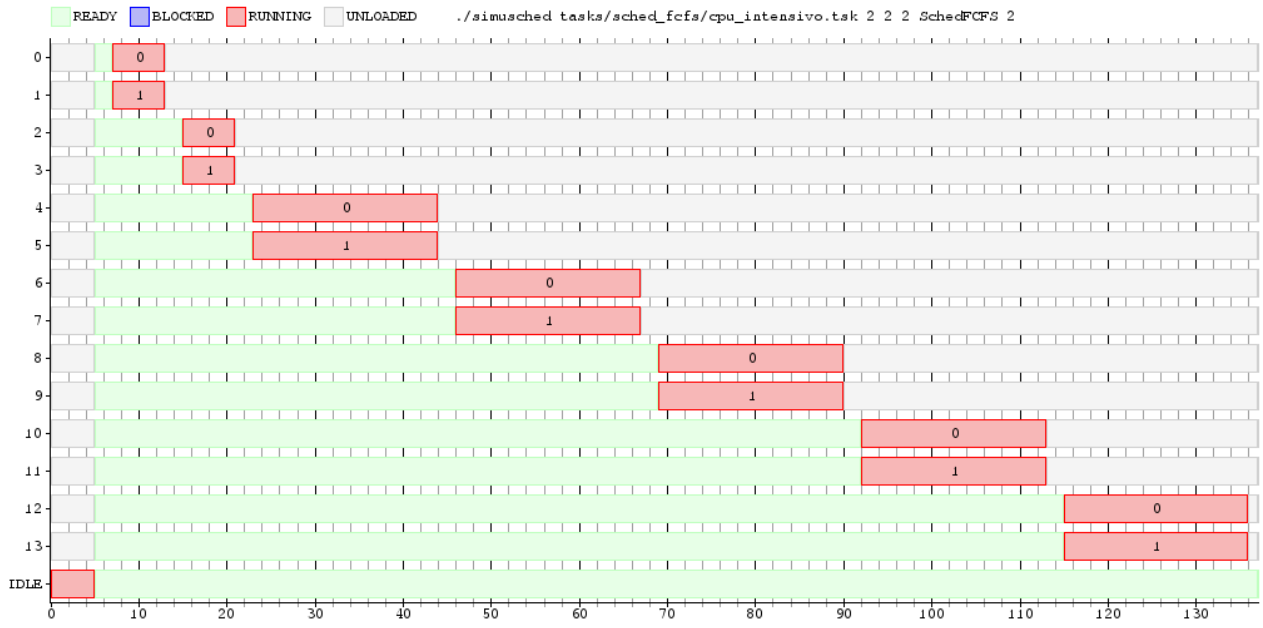

Figure 1.2.2: Intensivas en CPU

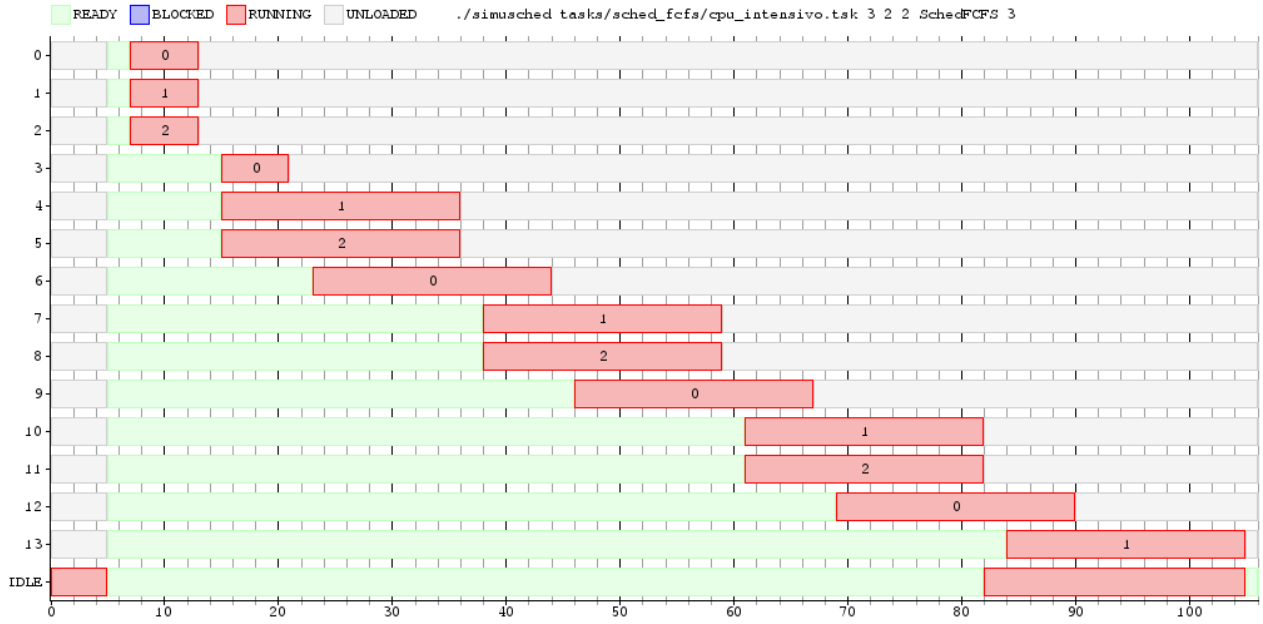

Figure 1.2.3: Intensivas en CPU

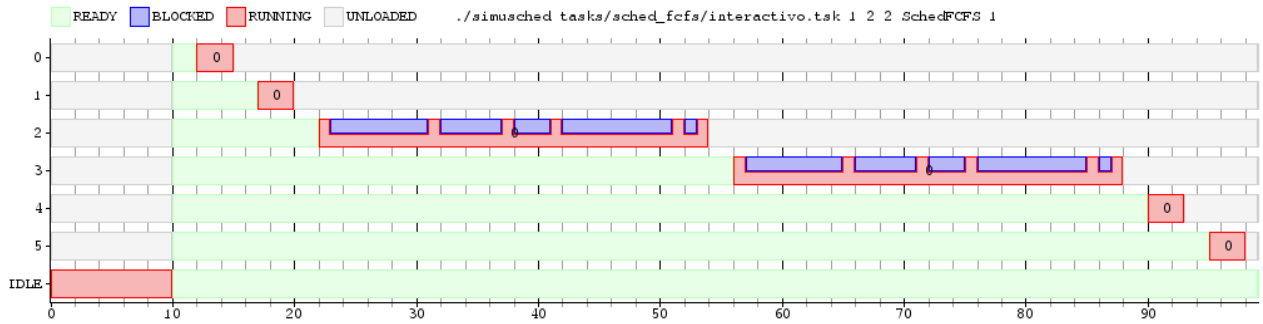

Figure 1.2.4: Interactivas

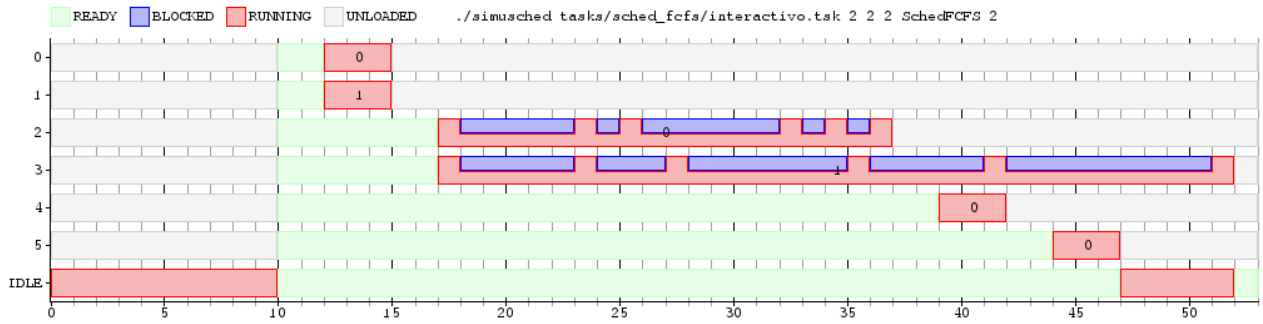

Figure 1.2.5: Interactivas

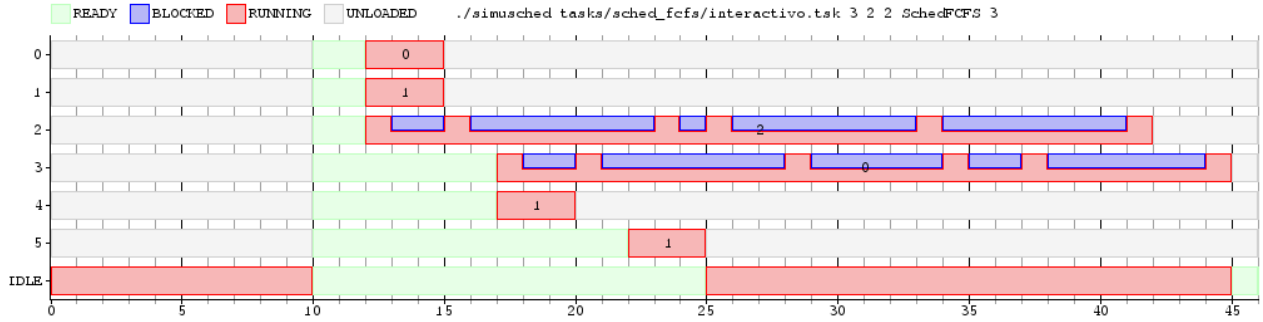

Figure 1.2.6: Interactivas

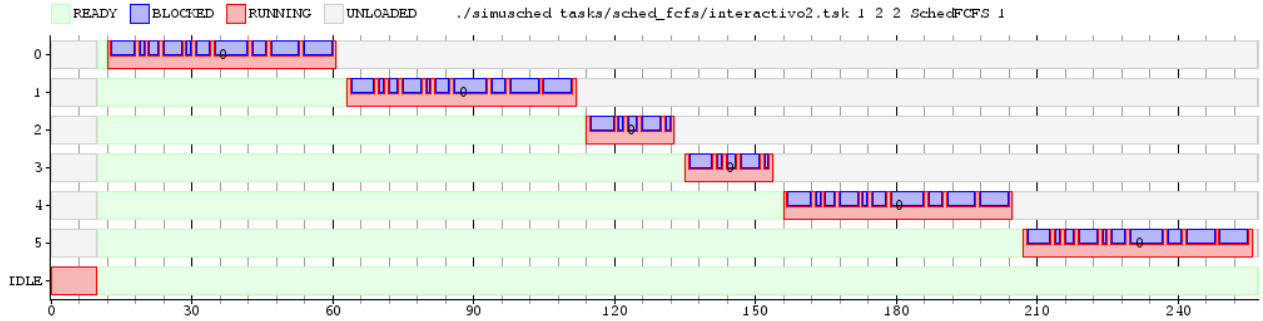

Figure 1.2.7: Interactivas (2)

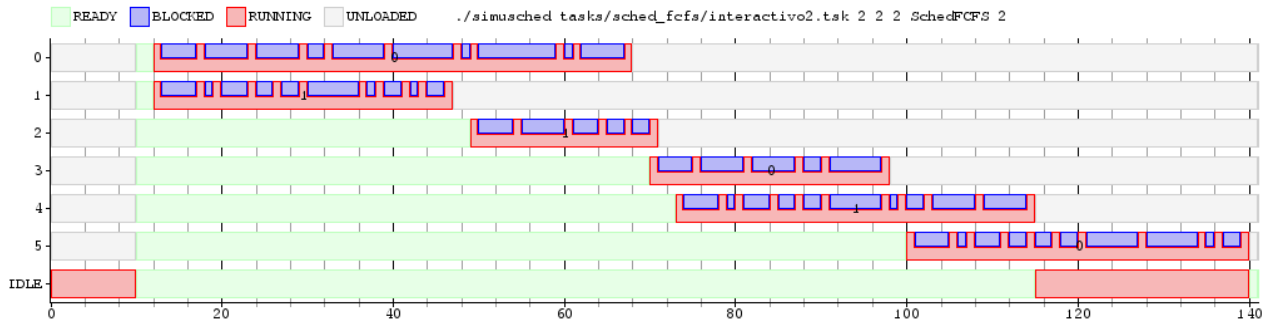

Figure 1.2.8: Interactivas (2)

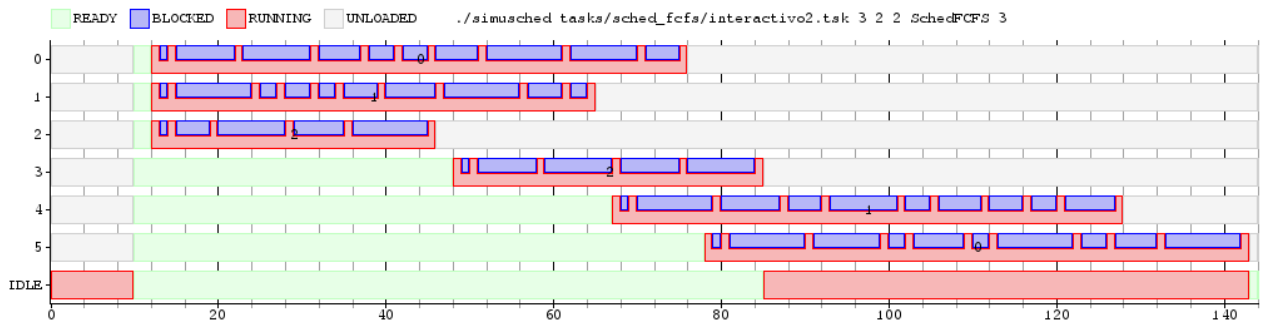

Figure 1.2.9: Interactivas (2)

Parte II Extendiendo el simulador con nuevos schedulers

2.3 Ejercicio 3

Ahora pasaremos a explicar algunos aspectos relevantes de la implementación que hicimos del shceduler $Round-Robin$, cuya idea básica es asignar espacios de tiempo equitativos a cada proceso y ir pasando de uno a otro en forma circular, sin asignar ningún tipo de prioridad a ningún proceso. Una característica interesante de este sistema de scheduling es que nos aseguramos de no tener el problema conocido como *starvation*.

En la implementación, recibimos como parámetro la cantidad de núcleos con la que vamos a trabajar y los valores de sus respectivos *quantums*. Utilizamos una única cola global, para permitir así la migración de proceso entre núcleos. La clase SchedRR consta de una parte pública y una privada. En la pública, se cuenta (al igual que en SchedLottery) con las funciones $load(pid)$, $unblock(pid)$ y $tick(cpu, motivo)$. La función de load es la de notificar al scheduler que un nuevo proceso ha llegado. Cada vez que esto sucede, agregamos a la cola de procesos el pid del de dicho proceso. Unblock se encarga de que en la próxima llamada a la función tick, el proceso pid esté disponible para ejecutar. Y finalmente, tick se encarga de manejar, de acuerdo a qué fue lo último que hizo el proceso antes de llamar a dicha función y a la porción del *quantum* ya utilizado por el proceso, de encolar en la cola de procesos al proceso actual y pasar a correr el próximo proceso si ya se utilizó todo el *quantum*, de pasar a correr el próximo proceso y dejar fuera de la lista de procesos al proceso actual si dicho proceso ha finalizado, o de poner a correr el próximo proceso y encolar en la lista de procesos al proceso actual si dicho proceso se ha bloqueado por alguna razón.

Por otro lado, esta la parte privada de la clase SchedRR. Consta de algunas variables que utilizamos para efectuar correctamente el manejo de procesos según la plítica del round-robin. Usamos un $vector < int > cores\_ticks$, con el que llevamos la cuenta en todo momento de la cantidad de ticks que consumió el proceso actual, para cada núcleo. También utilizamos un $vector < int > cores\_quantums$, en el cuál almacenamos los quantums de cada procesador que nos son pasados por parámetro. Otra estructura de datos importante es una $queue < int > process\_queue$, en la cuál almacenamos todos los procesos que están esperando a ser ejecutados. En $cores_count$ guardamos la cantidad de núcleos con la que vamos a trabajar. Finalmente, implementamos la función $run\_next(cpu)$, que devuelve el pid del próximo proceso de la cola a ejecutar. Sólo la invocamos cuando es necesario pasar a ejecutar un nuevo proceso. Y lo que hacemos es sacar el primer proceso que está esperando en la cola de la misma y ponerlo a correr, al mismo tiempo que inicializamos la variable $cant\_ticks[pid]$ a cero (inicializamos el contador de ticks para el proceso actual en ejecución a cero).

2.4 Ejercicio 4

Para este ejercicio se pide diseñar lotes de tareas, y luego simularlos mediante el Scheduler `SchedRR` implementado en el **punto 3**. Se diseñó un lote de **tareas simples**[2], y tres lotes conteniendo **tareas simples** y **tareas interactivas** con distintos criterios de orden. Se corrió el experimento con `1 CPU`, y con `2 CPU`, con un *Quantum* de 2.

simples.tsk

```
*10 TaskCPU 5
```

interactivas.tsk

```
*6 TaskCPU 6
*6 TaskConsola 6 1 12
```

interactivas2.tsk

```
*6 TaskConsola 6 1 12
*6 TaskCPU 6
```

interactivas_intercaladas.tsk

```
TaskCPU 6
TaskConsola 6 1 12
TaskCPU 6
TaskConsola 6 1 12
TaskCPU 6
TaskConsola 6 1 12
TaskCPU 6
TaskConsola 6 1 12
TaskCPU 6
TaskConsola 6 1 12
```

En el lote de **tareas simples**, estas comienzan al mismo tiempo, y son ejecutadas con los mismos parámetros, de forma tal que tanto en la simulación con `1 core` como en la de `2 cores` es fácil visualizar el comportamiento característico del **Round Robin**. Esto se puede apreciar en ***Figura 2.4.10*** y ***Figura 2.4.11***, de uno y dos **cores** respectivamente. En ambas versiones se puede notar que en ningún momento hay un "desperdicio" de CPU, pero dado que no hay llamadas bloqueantes esto es de esperarse. Por la misma razón, también es más notorio el orden en que son llamadas las tareas, ya que como estas jamás se bloquean, siempre están disponibles apenas terminan su *Quantum*. En la versión de **2 cores** se puede observar, además, que no se realiza migración de procesos entre núcleos, ya que el número de tareas es par. Si el número de tareas fuera impar, esto obligaría a todas las tareas a migrar de núcleo.

Los lotes de **tareas interactivas** producen resultados más interesantes. Para probar las diferentes opciones se varió el orden de las tareas. En ***Figura 2.4.12*** y ***Figura 2.4.13*** se pueden observar los diagramas correspondientes a las versiones de 1 core y 2 cores, para los lotes en donde se encolan primero las **6 tareas simples**, y luego las **6 tareas interactivas**. En la versión de **1 core**, vemos que no hay *desperdicio de CPU*. Sin embargo, en la versión de **2 cores**, se puede observar que la **tarea IDLE** corre varias veces, durante una cantidad considerable de ciclos, luego de que todas las **tareas simples** ya terminaron su ejecución. Esto coincide con los momentos en que todas las **tareas interactivas** se encuentran *bloqueadas*. También podemos notar que, aunque en la segunda versión el

[2]Solo usan CPU.

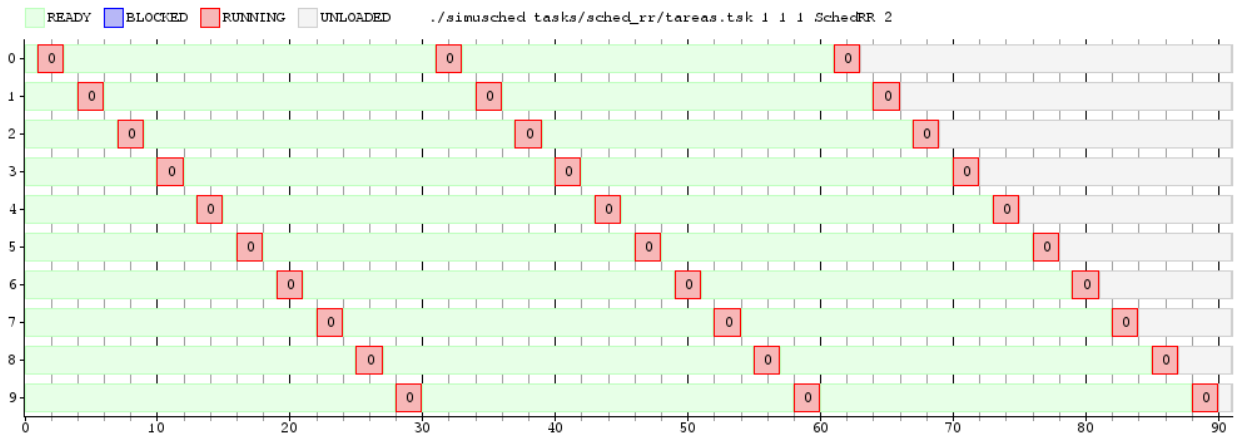

Figure 2.4.10: *Diagrama Gantt* de `SchedRR` para tareas simples **(1 core)**

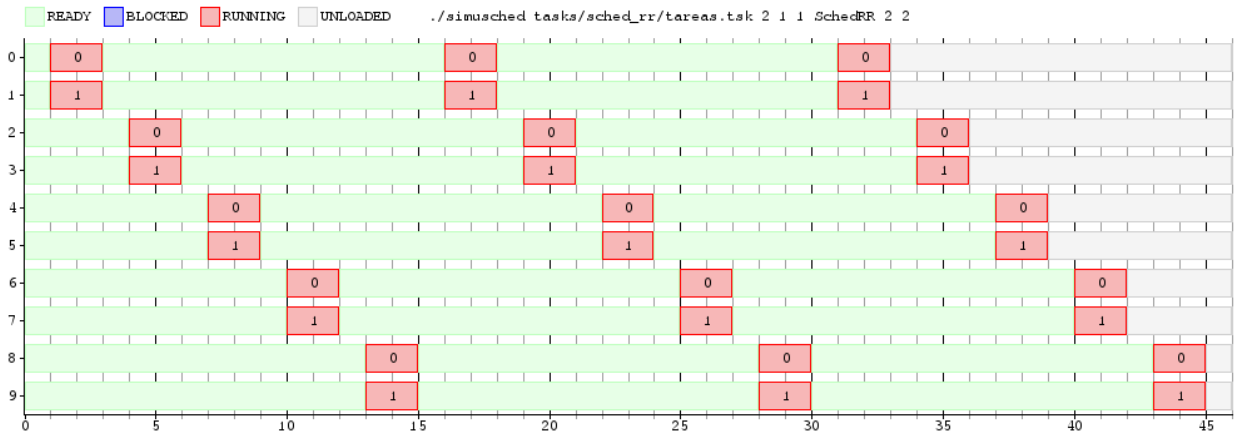

Figure 2.4.11: *Diagrama Gantt* de `SchedRR` para tareas simples **(2 cores)**

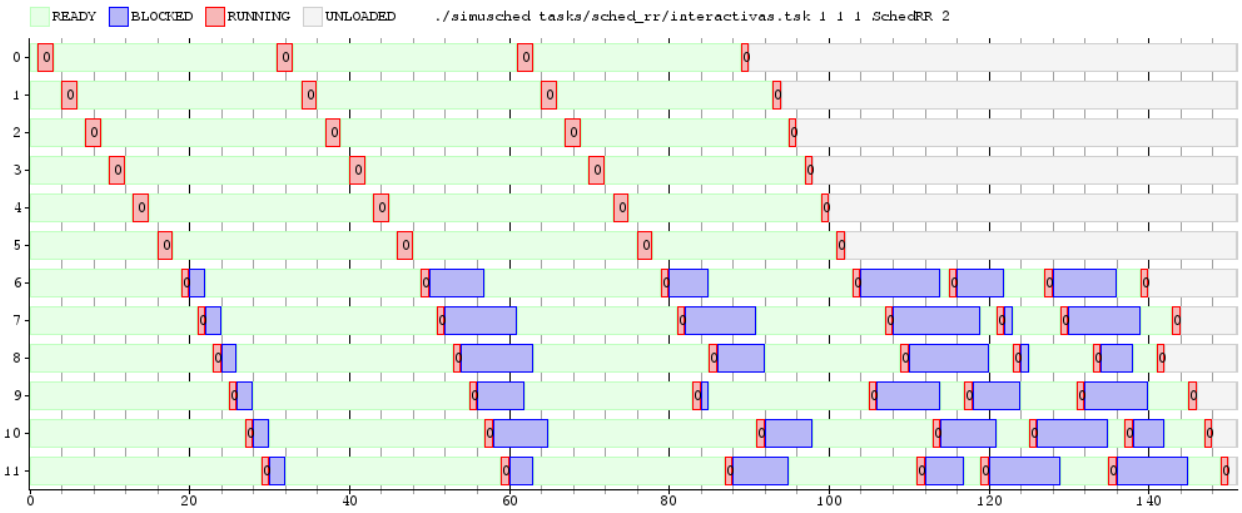

Figure 2.4.12: *Diagrama Gantt* de `SchedRR` para tareas simples, luego interactivas **(1 core)**

número de cores es el doble, el tiempo de ejecución total de todas las tareas no es la mitad que para la primera versión.

En ***Figura 2.4.14*** y ***Figura 2.4.15*** se pueden observar los diagramas correspondientes a las versiones de 1 core y 2 cores, para los lotes en donde se encolan primero las **6 tareas interactivas**, y luego las **6 tareas simples**. Pese al cambio de orden, se sigue notando el mismo tipo de patrón, característico de una planificación por **Round Robin**. En la versión de **2 cores** se nota una leve mejoría con respecto a la versión de ***Figura 2.4.13***, a causa de un mejor aprovechamiento de la CPU: como las **tareas interactivas** fueron encoladas primero, se le está dando una pequeña "ventaja/prioridad" frente a las **tareas simples**. Esto permite que cuando las primeras se bloquean, las segundas *aprovechen el tiempo de CPU* libre, evitando delegárselo a la **tarea IDLE**.

Finalmente, en ***Figura 2.4.16*** y ***Figura 2.4.17*** también se puede apreciar fácilmente el comportamiento de la *planificación Round Robin*. Las tareas se encuentran intercaladas, buscando así forzar una distribución más equitativa entre las **tareas interactivas** y las **tareas no interactivas**. En la versión de **1 core**, se puede notar rápidamente cómo el orden inicial se va perdiendo, relegando la ejecución de las **tareas bloqueantes**. Esto último se observa todavía más acentuado en la versión de **2 cores**, en donde además hay una alta tasa de migración de CPU. A pesar de ello, las **tareas no interactivas** tienen un tiempo total de ejecución levemente superior a ***Figura 2.4.13***, en donde estas son puestas al principio de la cola.

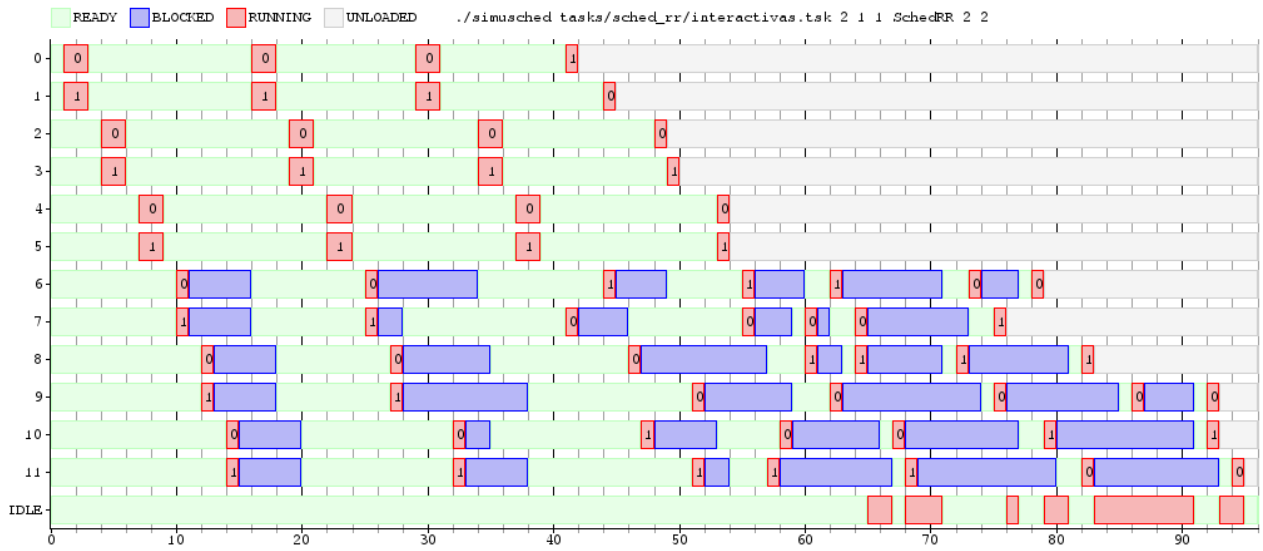

Figure 2.4.13: *Diagrama Gantt* de `SchedRR` para tareas simples, luego interactivas **(2 cores)**

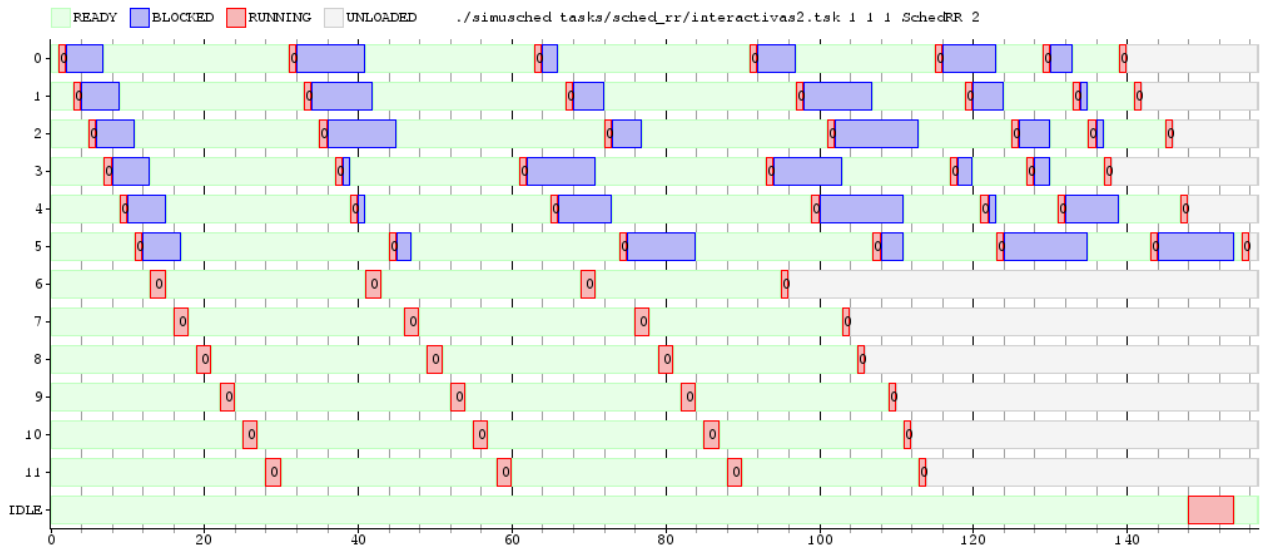

Figure 2.4.14: *Diagrama Gantt* de `SchedRR` para tareas interactivas, luego simples **(1 core)**

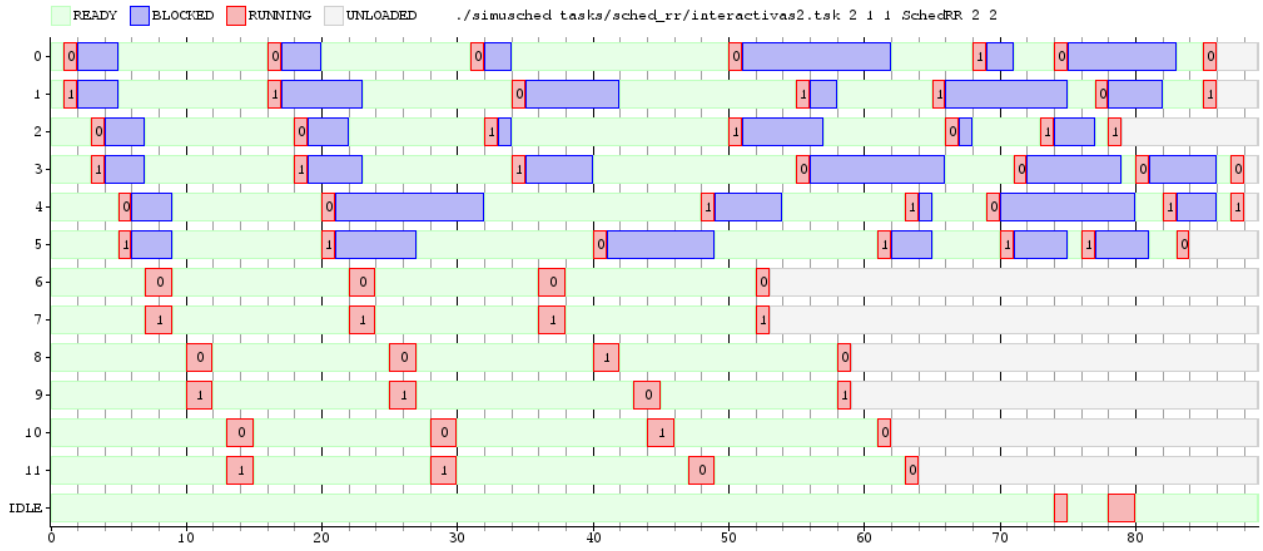

Figure 2.4.15: *Diagrama Gantt* de `SchedRR` para tareas interactivas, luego simples **(2 cores)**

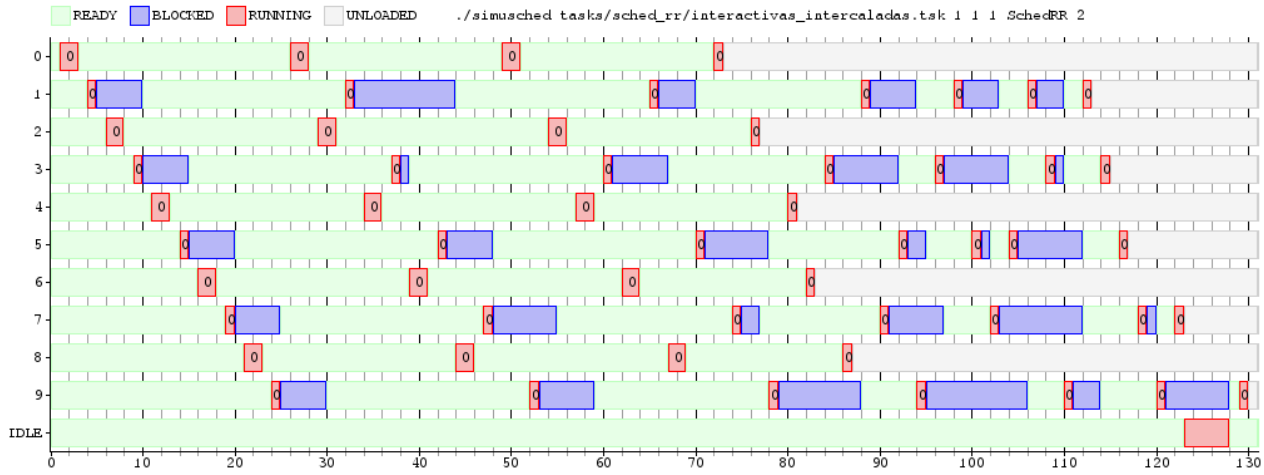

Figure 2.4.16: *Diagrama Gantt* de `SchedRR` para tareas simples e interactivas, intercaladas **(1 core)**

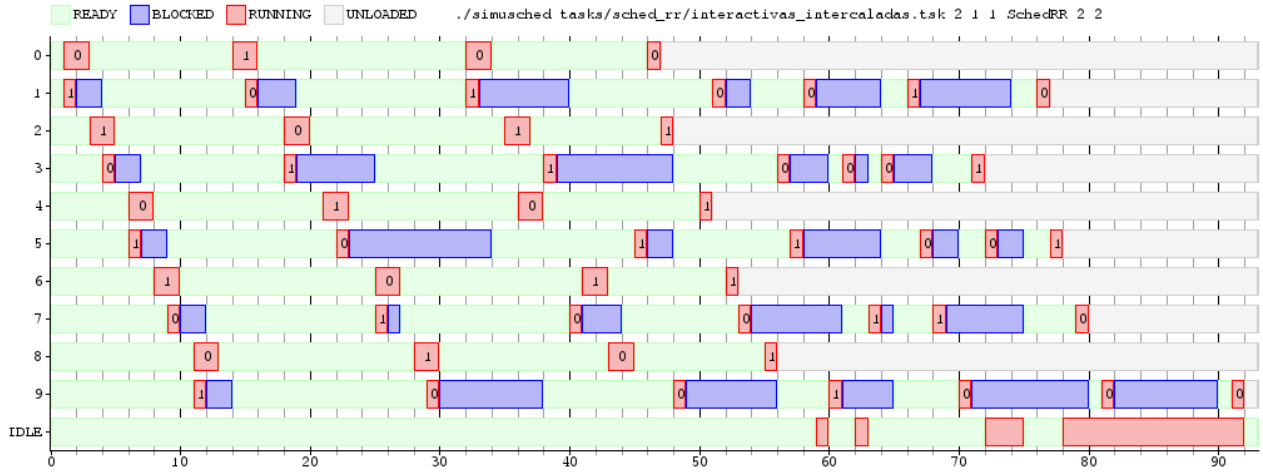

Figure 2.4.17: *Diagrama Gantt* de `SchedRR` para tareas simples e interactivas, intercaladas **(2 cores)**

2.5 Ejercicio 5

En este ejercicio, basados en el paper de Waldspurger, C.A. y Weihl en el cuál presentan un novedoso sistema de Scheduling llamado *LotteryScheduling*, implementamos una clase que llamamos SchedLottery que recibe como parámetros el quantum que se va simular y una semilla pseudo-aleatoria, que el scheduler va a utilizar para el manejo de los procesos. Como dice en el enunciado, básicamente nos interesó implementar la idea básica del algoritmo y la optimización de tickets compensatorios y en este caso, siempre trabajamos sólo con un núcleo.

La clase SchedLottery tiene como parte pública las funciones `load(pid)`, `unblock(pid)` y `tick(cpu, motivo)`.

i) La función load(pid) la utilizamos para notificar al scheduler que un nuevo proceso ha llegado. Cada vez que llega un nuevo proceso le damos un ticket a dicho proceso. De esta forma, en el próximo sorteo el proceso será candidato a ganarlo y así ser el próximo en ser ejecutado.

ii) La función tick(cpu, motivo). El parámetro motivo puede significar que una de tres cosas ocurrieron durante el último ciclo del reloj: la tarea pid consumió todo su ciclo utilizando el CPU número *cpu* (en cuyo caso incrementamos la cantidad de ticks y si esta cantidad supera el quantum de dicho procesador, la desalojamos), la tarea ejecutó una llamada bloqueante o permaneció bloqueada durante el último ciclo (en cuyo caso proveemos al proceso bloqueado la compensación de tickets correspondiente basándonos en el paper antes mencionado) ó porque la tarea pid terminó (en cuyo caso desalojamos la tarea actual). Después de hacer estos chequeos, corremos `run_lottery()` que se encarga de efectuar el sorteo de tickets para la próxima vez que se necesite elegir un proceso ganador y así asignarle los recursos necesarios.

iii) La función unblock(pid) hace que en la próxima llamada a la función tick, el proceso pid esté disponible para ejecutar. Lo que hacemos, es entregarle al proceso pid la cantidad de tickets que tenía el proceso antes de ser bloqueado multiplicado por la compensación correspondiente, que depende de qué fracción del quantum dicho proceso hizo del CPU.

Por otro lado, la clase SchedLottery tiene una parte privada, que consta de variables como *quantum*, que usamos para saber el quantum que tiene disponible cada proceso para ejecutarse, *seed*, que usamos como semilla para elegir el ticket ganador cada vez que hagamos un sorteo, $vector < ticket >$, que usamos para saber qué tickets corresponden a cada proceso, *total_tickets* para tener rápido acceso a la cantidad de tickets que hay distribuidos entre los procesos y *tick_number* que usamos para llevar la cuenta de cuántas veces llamamos a la función tick(cpu,motivo). Además, contamos con las funciones privadas $compensa(pid)$, $desaloja(pid)$ y $tickets\_index(pid)$. La primera se encarga de calcular y asignar la compensación deseada a un proceso cuando corresponda; la segunda de que cuando se bloquea un proceso, no esté disponible para ejecutar en la próxima ejecución de la función tick(cpu,motivo). Es decir, que la probabilidad de ser elegible en el sorteo sobre los tickets sea cero. Para eso, lo que hicimos fue sacarle todos los tickets temporalmente a dicho proceso.

Parte III Evaluando los algoritmos de scheduling

3.6 Ejercicio 6

Se registró un nuevo tipo de tarea, `TaskBatch(total_cpu, cant_bloqueos)`. Hace uso del CPU *total_cpu* ticks de reloj en total, entre los cuáles hace *cant_bloqueos* llamadas bloqueantes, distribuidas de manera uniforme y de duración un tick de reloj. La implementación requirió elegir *cant_bloqueos* momentos entre 1 y *total_cpu*. Para eso se guardó un arreglo de booleanos que mantiene que momentos ya fueron elegidos. Entonces se fueron generando momentos de manera pseudoaleatoria hasta elegir *cant_bloqueos* momentos *distintos* para hacer llamadas bloqueantes. En el resto del tiempo, la tarea usa el procesador.

3.7 Ejercicio 7

3.8 Ejercicio 8

En esta sección implementamos una segunda versión para el scheduler tipo $Round - Robin$ que ya implementamos en ejercicios anteriores. En este caso, el objetivo es no permitir la migración de procesos entre núcleos y analizar qué sucede con la performance de este scheduler en comparación con el original para distintos lotes de tareas. En principio, esperaríamos que en algunos tipos de lotes un algoritmo ande mejor que el otro Esto es porque, por ejemplo, pensamos que si tenemos un único proceso que tarda mucho en completarse y no permitimos migración entre procesos, usando el método alternativo empeoraríamos la performance mientras que usando el otro método, el tiempo total que se tardaría en finalizar el proceso se reduciría a la mitad. Sin embargo, si aparte de dicho proceso largo tuviéramos un segundo proceso igual de largo, la implementación más eficiente sería la que no permite migración entre núcleos (así nos ahorraríamos de perder tiempo en cambios de contexto). Finalmente, en principio creemos que si hubieran muchos más procesos que núcleos y de longitud similar, y no se permitiera la migración de procesos entre núcleos, los tiempos de finalización de distintos procesos serían muy dispares. Algo que, en principio es indeseable pues podría suceder que si a un proceso muy corto le toca justo el mismo core que a un proceso muy largo, este debería esperar mucho tiempo a que el largo termine hasta que el procesador lo atienda.

En nuestra implementación del la clase $SchedRR2$ para implementar las mismas tres funciones básicas que venimos implementando en los anteriores schedulers (tick, unblock, load), utilizamos en la parte privada de la clase tres estructuras para el manejo de los procesos: estas son un $vector < core >$, que contiene mucha de la información que teníamos en la implementación del $SchedRR$(en efecto, tenemos almacenado el quantum del core, la cantidad de ticks actual, un vector de entero que lleva cuenta de los procesos activos en dicho core, una cola de esos procesos para saber en qué orden ir ejecutándolos, y una función runNextProcess() que se encarga de sacar de la cola al primer proceso y ponerlo a correr), pero en este caso efectuamos una distinción entre qué procesos están en cada núcleo (antes, no nos importaba hacer esta distinción, porque como había migración de procesos entre núcleos, no nos importaba en qué núcleo estaba cada proceso).

Hacemos uso de esta estructura a través de la función $getProcessCores(pid)$, que para el proceso pid nos devuelve un puntero al núcleo en el cuál está presente. Entonces, ahora en cada tick del reloj de cada cpu, si el motivo es de bloqueo o terminación, entonces pasamos a correr en ese mismo cpu al próximo proceso. Mientras que si llega un nuevo proceso, lo agregamos a la cola de procesos del cpu en el cual se efectuó el tick y no en ninguno otro.

3.9 Ejercicio 9

Se procedió a analizar la ecuanimidad o $fairness$ de nuestra implementación de *Lottery Scheduler*.

Se eligieron cuatro procesos idénticos: $TaskCPU$ con 30 ticks de uso del procesador. Se lanzaron al mismo tiempo y se midió la cantidad de ticks que tomaron para finalizar la ejecución. Se esperaba que los cuatro procesos terminaran al mismo tiempo, si esperábamos un mecanismo de scheduling ecuánime.

Sin embargo, al correr el simulador distintos procesos terminaron en tiempos con hasta un 40% de diferencia. Dado el carácter pseudoaleatorio de la elección del proceso a ejecutar, dada una corrida en particular no está garantizada una asignación equitativa de los recursos. Se extendió el experimento y se llegó a dos conclusiones importantes.

Primero, la ecuanimidad tiende a aumentar a medida que el tiempo aumenta. Se lanzaron 10 procesos idénticos al mismo tiempo, con un uso del CPU de 20 ticks, y un quantum de 4. Se fue midiendo, cada vez que se llama a `run_lottery()` y se elige un proceso, la cantidad de ticks de procesador otorgados a cada proceso hasta el momento. Como medida de ecuanimidad se calculó la desviación estándar de estos valores.

A continuación se presentan los resultados.

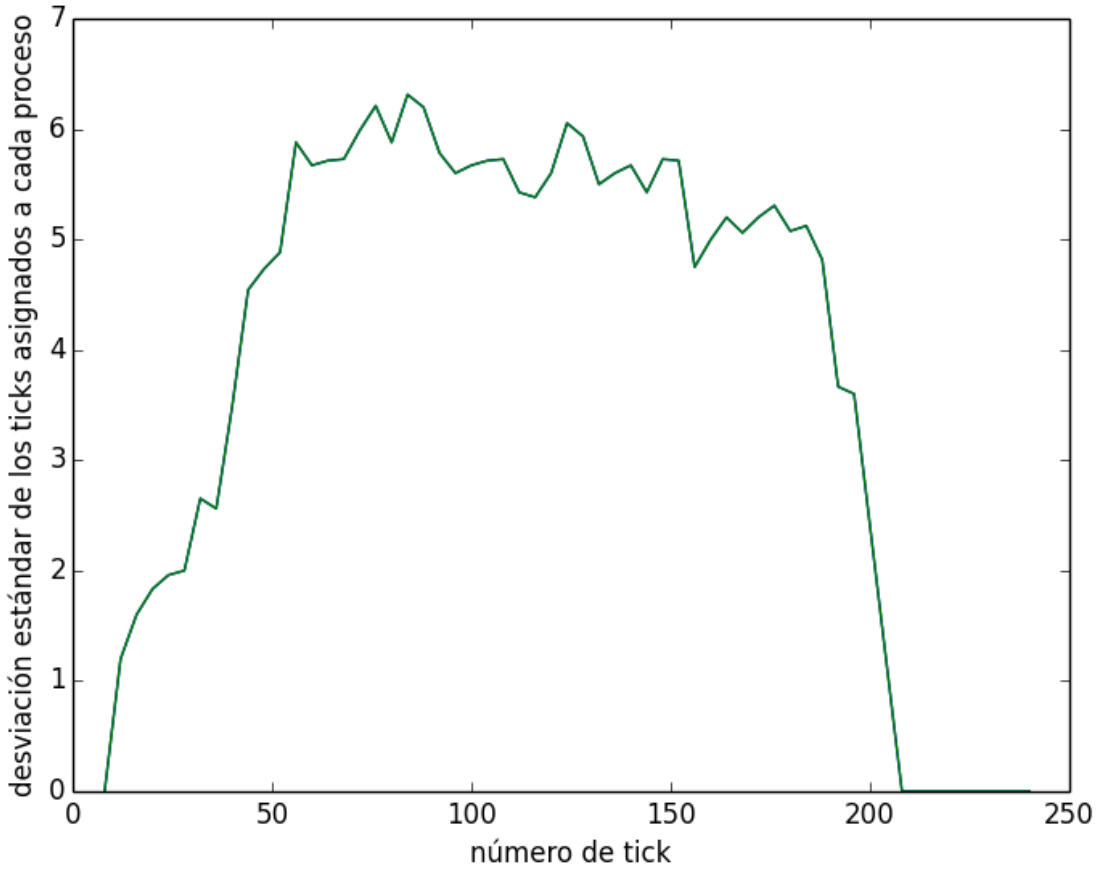

Figure 3.9.18: Desvío estándar de la cantidad de ticks asignados a cada proceso

Al comienzo, cuando todas los procesos corrieron por cero ticks de reloj, la desviación estándar vale cero. Cuando se hacen los primeros sorteos de proceso a ejecutar, la desviación aumenta abruptamente. Sin embargo, luego de estabilizarse comienza a bajar de forma constante a medida que sucesivos sorteos emparejan los ticks asignados a cada proceso. La pronunciada caída final se debe a que algunos procesos terminaron y fueron desalojados, por lo que los procesos que quedan notan una *inflación* en sus tickets, es decir, valen más ya que la cantidad de tickets en juego disminuye. La desviación estándar culmina en cero naturalmente ya que todos los procesos terminan con 20 ticks de CPU acumulados.

Luego, se extendió el experimento inicial midiendo el tiempo de ejecución de los cuatro procesos, ahora promediado a través de varias corridas. El procedimiento utilizado fue calcular el promedio de las primeras 1, 2, ..., 500 corridas. Los resultados se pueden ver a continuación.

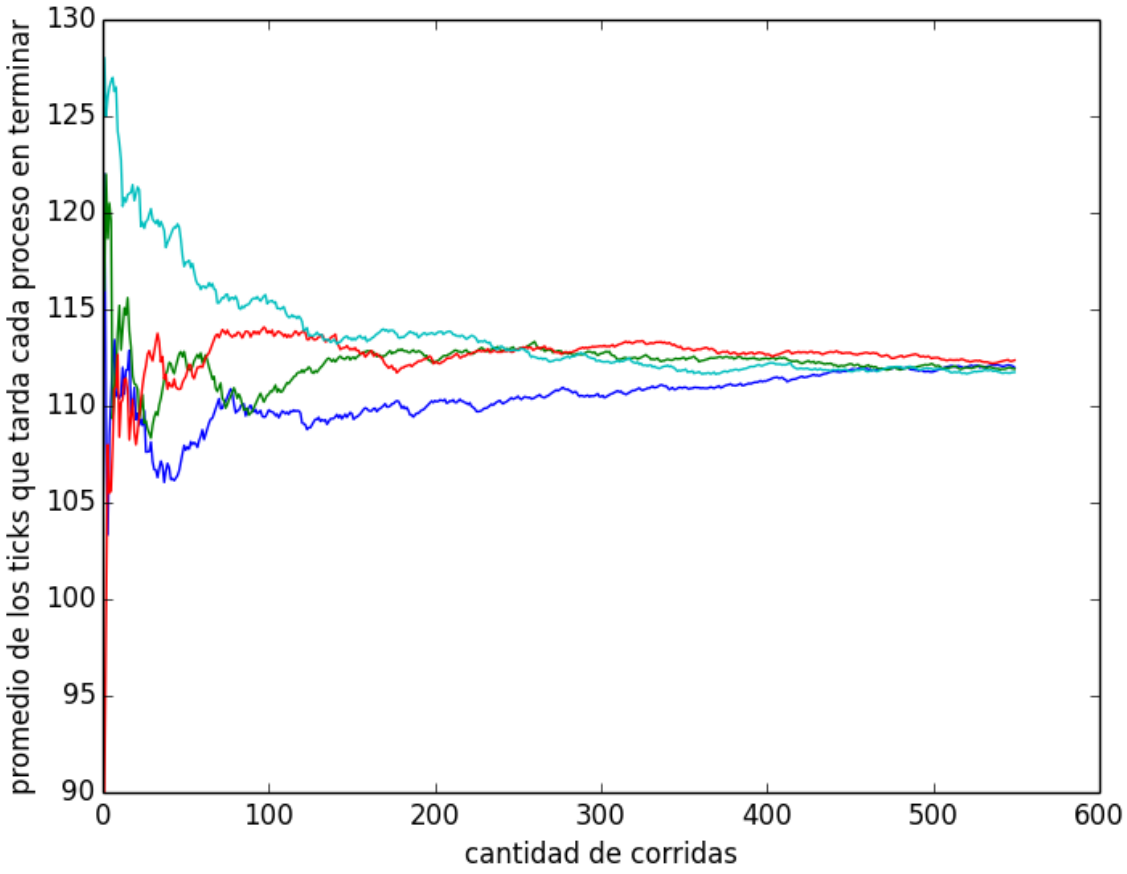

Figure 3.9.19: Tiempo de ejecución de cuatro procesos idénticos

Los tiempos de ejecución arrancan muy dispares. Primero se acercan hacia un valor intermedio de manera exponencial, luego van convergiendo lentamente hacia el mismo valor, en éste caso aproximadamente 112. Se puede concluir de forma fehaciente que a medida que la cantidad de corridas aumenta, la ecuanimidad del LotteryScheduler se acerca a la perfección.

3.10 Ejercicio 10

Los *compensation tickets* están diseñados para compensar un proceso que no llegó a completar su quantum al ser bloqueado, por ejemplo. Es un mecanismo sencillo de implementar en un *LotteryScheduler* pero que puede ser poco factible en otros schedulers. En la siguiente experimentación se puso a prueba que papel toma este mecanismo en garantizar la ecuanimidad del scheduling.

Se planteó el siguiente escenario:

- 5 procesos `TaskCPU(30)`
- 5 procesos `TaskAlterno(0, 100, 30)` es decir, se bloquean por 100 ticks y luego usan el CPU por 30 ticks
- quantum de 10 ticks
- todos los procesos lanzados al mismo tiempo

El interrogante que se plantea es si los procesos que hacen las llamadas bloqueantes van a terminar su ejecución mucho después que los procesos que solo usan el CPU. Se corrió el experimento sin valerse de los *compensation tickets* y luego haciendo uso de ellos.

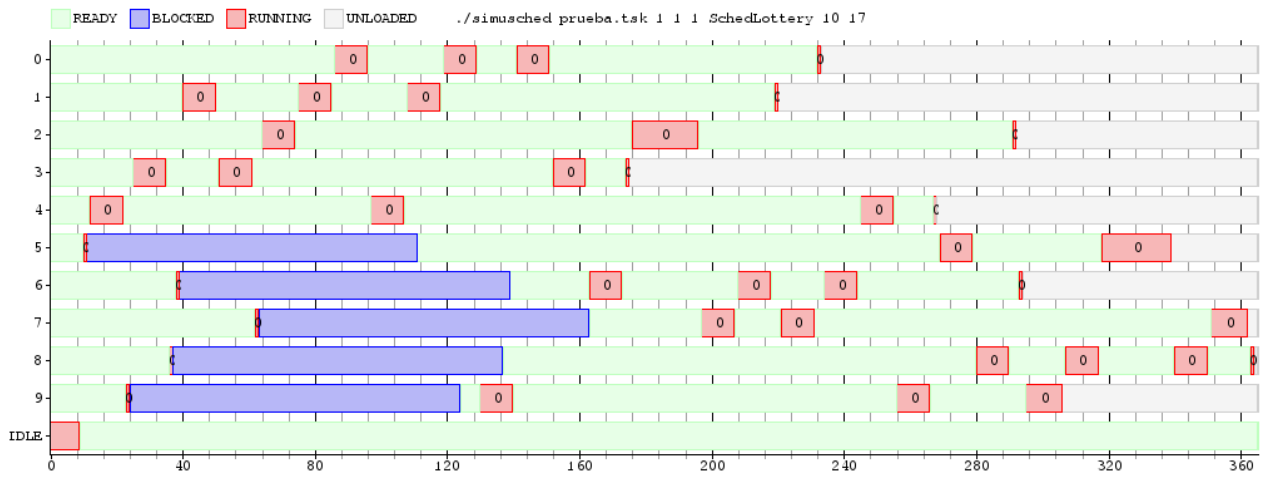

Figure 3.10.20: Diagrama de Gantt sin *compensation tickets*

Como es de esperar, mientras los procesos de IO se bloquean, el resto es elegido con frecuencia y puede correr gran parte de sus 30 ticks de CPU. Cuando los procesos de IO se desbloquean, el tiempo de CPU se divide equitativamente entre todos, pero, como los procesos de CPU ya habían corrido bastante tiempo previamente, terminan antes.

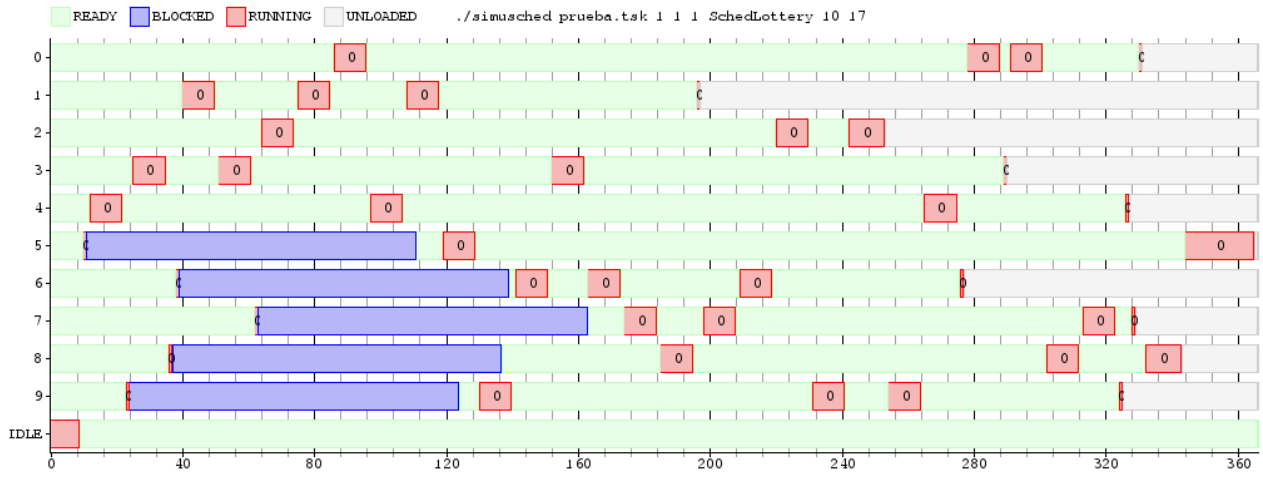

Figure 3.10.21: Diagrama de Gantt con *compensation tickets*

En este caso la ejecución comienza igual. Sin embargo, al desbloquearse los procesos de IO, se puede percibir otra tendencia. Como para realizar la llamada bloqueante utilizaron solo un tick de procesador, 1/10 del quantum asignado, se les entregan 10 tickets en vez de 1. En la siguiente ronda de loterías, estos procesos ejecutan casi como si estuvieran solos. Tienen 5/55 tickets cada uno contra 1/55 de los procesos de CPU. Al salir seleccionados, vuelven a tener 1 ticket cada uno y el resultado del lottery se empareja. Como consecuencia los procesos terminan más cerca unos de otros.

Este resultado se puede cuantificar en el siguiente gráfico.

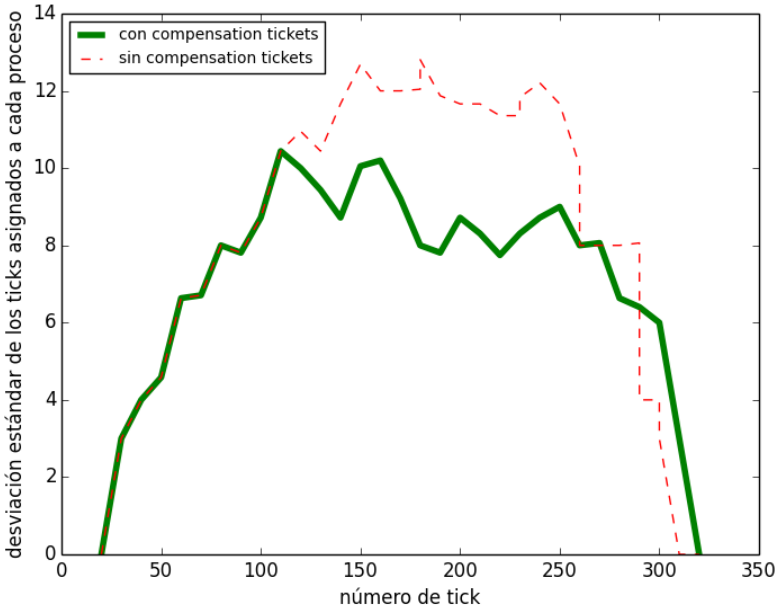

Figure 3.10.22: Comparación de *fairness* con y sin *compensation tickets*

La evolución del desvio estándar comienza igual, hasta el tick 120 donde se liberan los procesos bloqueados. A partir de ese momento la corrida con *compensation tickets* presenta una desviación observablemente menor. Lo que se concluye es que los *compensation tickets* son un mecanismo que se puede implementar fácilmente en el *LotteryScheduler*, que no implican un *overload* excesivo en el scheduling y que permiten aumentar corroboradamente la ecuanimidad en la asignación de recursos.