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	201/12	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.

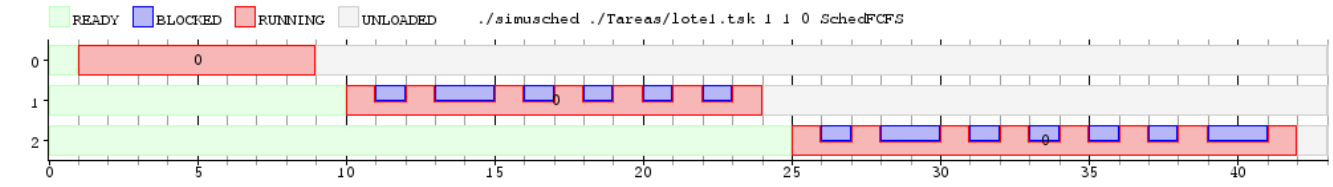
1.2. Ejercicio 1

El objetivo de este ejercicio es implementar una tarea de tipo **TaskConsola**, 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*.

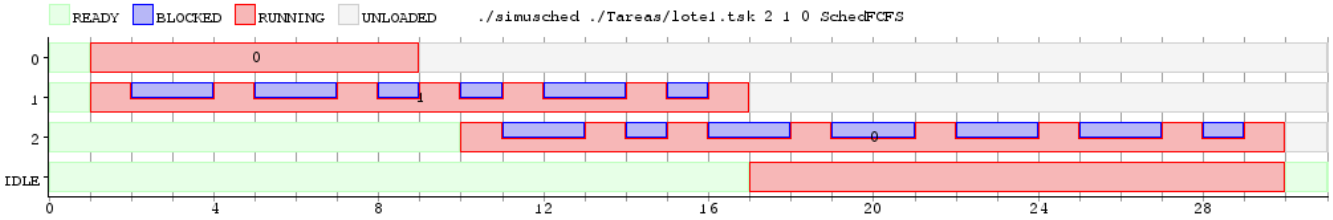
1.3. Ejercicio 2

El objetivo de este ejercicio es ejecutar un lote de tareas, una intensiva en CPU y las otras 2 de tipo interactivo (**TaskConsola**), 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.

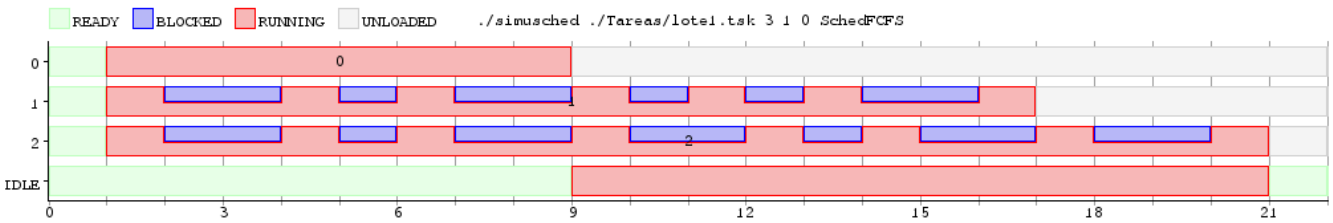
El lote de tarea que utilizamos es el *lote1.tsk*. A continuación se pueden ver los gráficos:



1 core.



2 cores.



3 core.

Los gráficos nos están indicando que a mayor cantidad de cores, mejor es el rendimiendo de la política FCFS, tanto si tomamos cómo métricas el *waiting time* como el *turnaround* de las tareas. En caso de tareas de mucha actividad I/O esta política no es eficiente, ya que se desperdicia CPU cuando esta espera que la tarea se desbloquee, en lugar de ejecutar otra.

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**, 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 **BLOCK** o **EXIT**, sencillamente se devuelve la proxima tarea en la cola, sin encolar nuevamente la tarea actual, y se pone el contador de ticks para ese cpu en 0. En caso de no haber más tareas, se ejecuta **IDLE_TASK**. Cuando un proceso llama a la función `unblock`, esta vuelve a encolar el proceso en la cola de tareas.

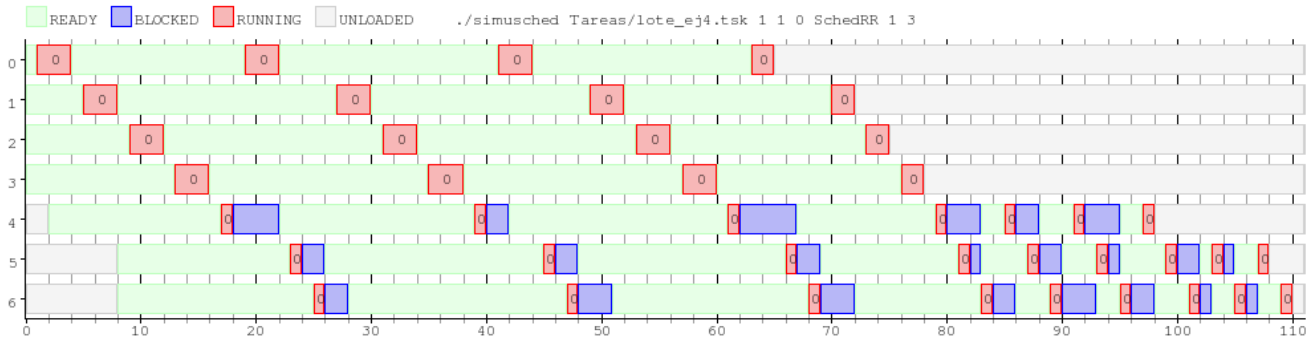
Para más detalles, consultar la implementación en el archivo *sched_rr.cpp*.

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 cuan 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 mas en el momento 8.

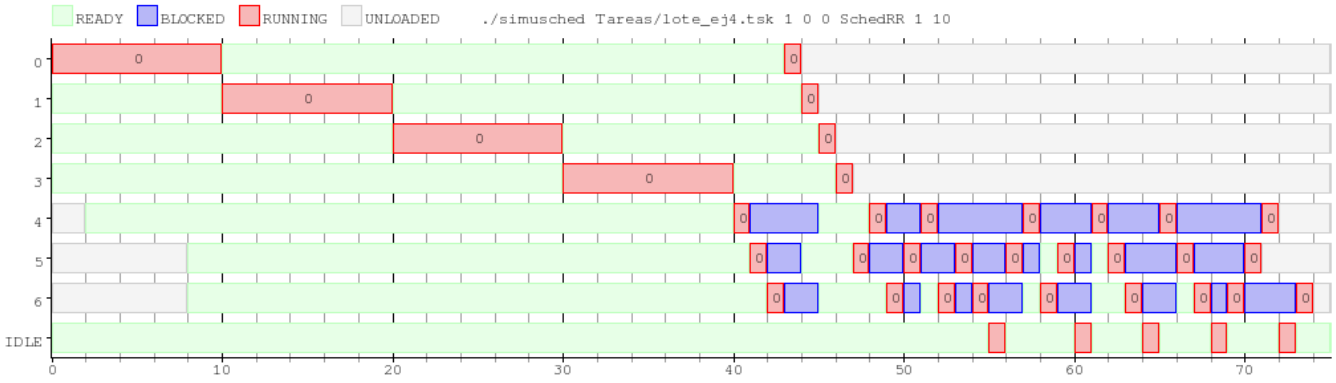
Como primer política vamos a tomar un RR con quantum 3, costo de cambio de contexto de 1 y un sólo un core. El procesamiento de dicho lote arroja el siguiente gráfico:



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

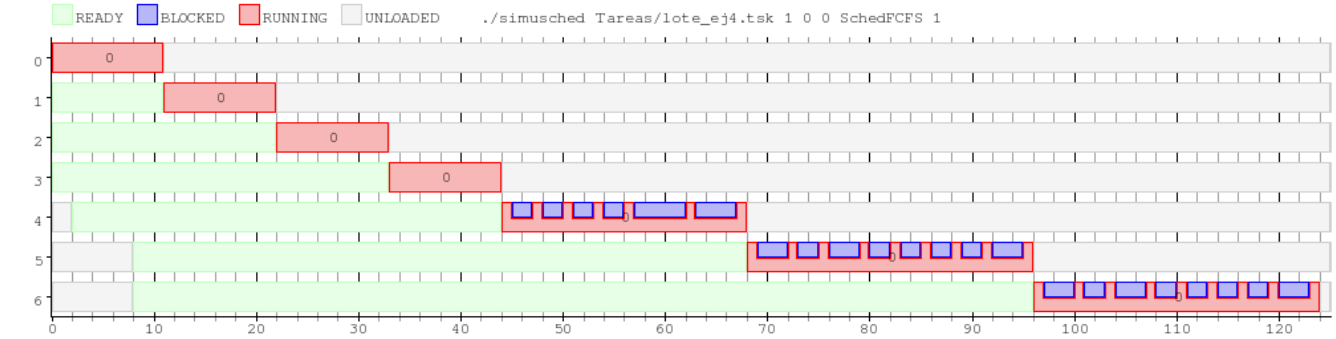
Como podemos apreciar, se ejecutan en una primera pasada las tareas de cpu (de la 0 a la 3 inclusive) que no tienen bloqueos. Luego se ejecuta la tarea 4, que llegó en el momento 2. Luego en lugar de ejecutarse la 5 o la 6 (que llegaron en el momento 8) vuelve a ejecutarse la tarea 0, luego la 1 y recién 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 primer 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 expermiento con un quantum de 10.



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

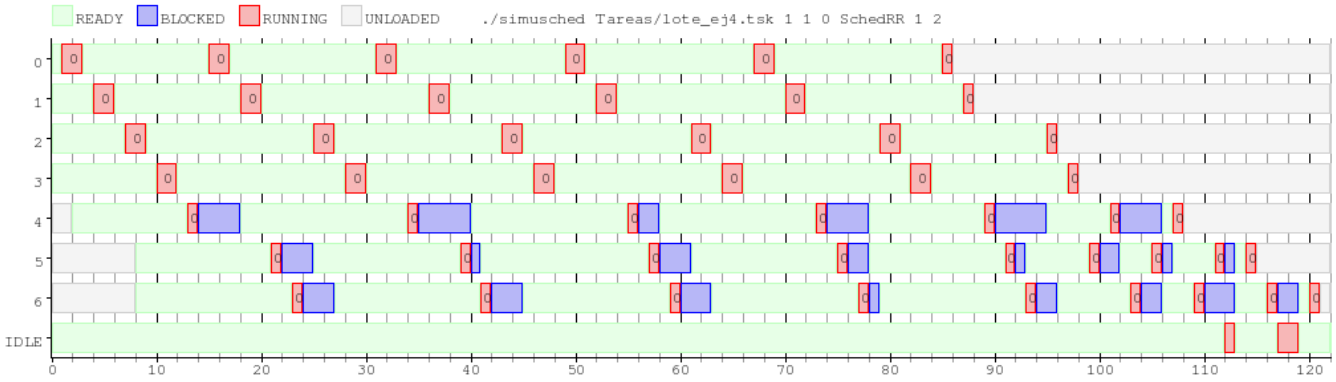
Y a continuación el mismo lote de tareas procesado con FCFS.



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 entorpecerÃa 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.

2.4. Ejercicio 5

Para el ejercicio 5 se nos pidió diseñar e implementar un algoritmo de *scheduling* basado en la publicación de Waldspurger, C.A. y Weihl, W.E llamada *Lottery scheduling: Flexible proportional-share resource management.*

2.4.1. Introducción

El algoritmo de *Lottery scheduling* establece un sistema de prioridad para la obtención de los recursos del sistema por parte de los procesos, basado en *tickets* de lotería. Cada proceso es dueño de una determinada cantidad de tickets. En cada elección de un nuevo proceso a correr por parte del scheduler, se realiza una lotería entre todos los tickets del sistema, eligiéndose uno al azar. El próximo proceso a correr es el dueño del ticket ganador, quien se ejecutará hasta la finalización del *quantum*, y el proceso se repite.

2.4.2. Asignación de recursos

La distribuciíon de los tickets y posterior elección aleatoria de uno de ellos establece un sistema de prioridades probabilístico basado en la cantidad de tickets que posee cada proceso. Por ejemplo, sean A y B dos procesos en un sistema de 100 tickets, teniendo A 75 tickets y B 25, se esperaría que el proceso A posea el 75 % de los recursos del sistema y B el 25 %. Dado el carácter no determinístico del algoritmo, no es posible confirmar que esto suceda para cualquier ejecución de las tareas. Pero la probabilidad de que esto suceda aumenta conforme a la cantidad de ejecuciones, como se mostrará en la sección de experimentación. Dada que la asignación de recursos esperada es proporcional a la cantidad de tickets que posee cada proceso, decimos que *Lottery Scheduling* es probabilísticamente justo.

Los tickets *encapsulan* la gestión de la asignación de los recursos del sistema, ya que cuantifican la posesión de éstos por parte de los procesos independientemente de los detalles de la máquina. Dado que en un determinado sistema puede haber diversos recursos heterogéneos, realizan una abstracción uniforme de éstos ya que la probabilidad de asignación de cualquiera de ellos a un proceso se representa homogéneamente a través de tickets.

Esta representación facilita también los cambios de prioridad entre los determinados procesos.

Nota: Dado que el único recurso de sistema que se gestiona en nuestra implementación del scheduler es el tiempo de CPU, de ahora en más solo nos referiremos a éste.

2.4.3. Optimizaciones

El concepto básico de *Lottery Scheduling* puede mejorarse considerablemente mediante la implementación de algunos mecanismos, explicados a continuación:

- *Transferencia de tickets:* cuando un proceso está bloqueado esperando la respuesta de otro proceso, puede transferir sus tickets a éste, causando una mejora de perfomance ya que va a tener más probabilidad de obtener el CPU, agilizando la respuesta. Cuando el primer proceso recibe la respuesta, le son devueltos sus tickets. De esta forma se resuelve el problema de la proridad inversa, de una manera similar a herencia de prioridades.
- *Tickets compensatorios:* los procesos que se bloquean en espera de una respuesta externa y no consumen el total de su quantum terminan obteniendo menos tiempo de CPU del que les corresponde, rompiendo el modelo de asignación en cuanto a la probabilidad. La forma de solucionar esto es la asignación de tickets compensatorios a estos procesos. De esta forma cuando un proceso utiliza una fración de su quantum, recibirá tickets aumentando su probabilidad de ser elegido para el quantum siguiente a su desbloqueo.
- *Inflación de tickets:* cuando un proceso sabe que debe ejecutar una sección crítica y considera que va a necesitar más tiempo de CPU, puede aumentar su número de tickets. Esta es una forma simple y eficiente de reflejar esa necesidad en el sistema, y no es necesario comunicarse con los otros procesos. Cabe aclarar que, este método sólo es utilizable en un sistema donde los procesos ejecutan de forma cooperativa.

2.4.4. Resumen: Ventajas de Lottery Scheduling

- Implementación sencilla y eficiente.
- Provee encapsulación abstracta, relativa y uniforme de los recursos.
- El usuario puede realizar la distribución de tickets entre sus procesos, estableciendo prioridades.
- Realiza una distribución de los procesos probabilísticamente justa.
- La randomización es una elección rápida, fácil de implementar, independiente de las anteriores y libre de peores casos.
- Se puede especificar qué porcentaje de procesador le corresponde a cada proceso.

- Se resuelve el problema de la inanición. Siempre y cuando un proceso tenga al menos un ticket, tiene una probabilidad no nula de ser elegido.
- Gracias al sistema de *currency*, puede implementarse un sistema modular de gestión de recursos.
- Mediante el concepto de *ticket transfer*, un proceso bloqueado esperando respuesta de otro proceso puede transferir sus tickets a éste, solucionando el problema de la prioridad inversa.

2.4.5. Posibles desventajas

- Si bien la naturaleza aleatoria del algoritmo tiene ventajas, posee como desventaja nunca poder asegurar determinísticamente un resultado.
- La asignación de los tickets a los procesos es un problema en sí mismo no abarcado por los autores. El comportamiento de un sistema va a depender fuertemente de la distribución de los tickets.

2.4.6. Detalles de la implementación

A continuación expondremos los detalles de nuestra implementación particular de *Lottery Scheduling*. Dado que la probabilidad de ser asignado el CPU para cada proceso depende solamente de la cantidad de tickets que posea, y no de cuáles sean éstos, no diferenciaremos entre tickets, sólo nos va a interesar la cantidad de tickets que posea cada proceso.

El sistema tiene una cantidad de 100 tickets que son distribuidos equitativamente entre todos los procesos , para garantizar la misma probabilidad a todos. Nuestra elección de 100 como número de tickets se debe a que el cálculo de la probabilidad de cada proceso en base a su número de tickes es simple e inmediato, y nos provee de presición suficiente. Es importante también notar que en la experimentación, no habrá más de 100 tareas ejecutándose en un mismo experimento. De todas formas la solución a esto no es más que cambiar la variable por un número mayor, por ejemplo 1000.

La estructura de datos elegida para almacenar las tareas listas y su cantidad de tickets correspondiente es una lista de pares$< pid, \#tickets >$.s La elección de esta estructura de datos se basa en la necesidad de recorrerla de principio a fin, y como se verá más adelante, que mantenga un orden particular.

Estructuras de datos:

- *int* quantum: almacena el valor de quantum pasado por parámetro.
- *int* cantTicks: almacena la cantidad de ticks de reloj que pasaron desde el último cambio de tarea.
- *int* semilla: almacena la semilla pseudoaleatoria
- $list < pid, cantTickets >$ tareasYTickets: lista que almacena las tareas *ready* del sistema y para cada una, la cantidad de tickets que posee

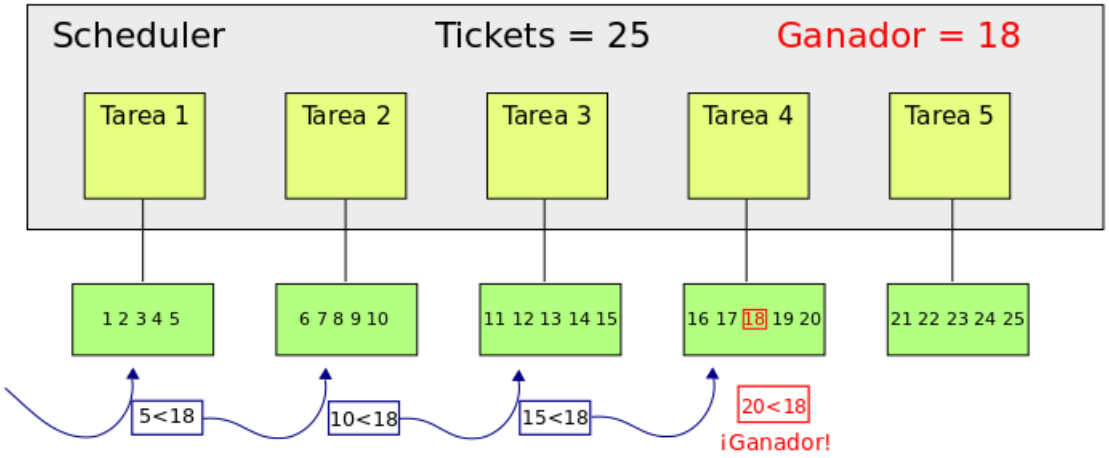
Nuestra clase scheduler recibe como parámetros el quantum y la semilla pseudoaleatorioa. *SchedLottery* hereda de la clase *SchedBase*,y por lo tanto implementa las funciones *load()*, *unblock()* y *tick()*, además cuenta con:

Funciones

- *load*() : carga una nueva tarea que llega al scheduler, agregándola a la lista, y repartiendo nuevamente los tickets entre todos los procesos de forma equitativa.
- *unblock*() : desbloquea una tarea, agregandola nuevamente a la lista de tareas ready y reparte nuevamente los tickets.
- *tick*() : se ejecuta en cada tick del reloj, incrementando en 1 la cantidad de ticks de la tarea actual. En caso de que se agote el quantum de ésta, se desaloja y se llama a lotería. Si la tarea se bloqueó terminó, se elimina de la lista de tareas ready y se llama a lotería. En caso de que no queden más tareas a ejecutar, finaliza la ejecución pasando a la tarea IDLE_TASK.
- *loteria*() : se realiza la lotería, eligiendo la nueva tarea a ejecutar. Se explica a continuación.

2.4.7. Lotería

Como mencionado anteriormente, nuestra implementación no diferencia entre tickets, solamente se tiene en cuenta la cantidad de ellos que cada proceso posee. La forma de elegir al ticket ganador, es elegir un número pseudoaleatorio entre 0 y la cantidad total de tickets. Acto seguido, se recorre la lista de tareas, y se realiza una suma parcial de la cantidad de tickets de las tareas recorridas. Se itera por las tareas, mientras la suma parcial sea menor al número ganador. La tarea ganadora va a ser aquella en la cual la iteración termine, es decir, la primera tal que la suma parcial + la cantidad de tickets de la tarea ganadora supera el numero pseudoaleatorio elegido.



Dado que los tickets poseen todos el mismo valor de probabilidad, y son uniformes, de esta forma se produce una numeración automática en cada lotería realizada, donde si cada tarea posee k tickets, entonces sus tickets son los del intervalo de números enteros $[sumaParcial\ ..\ sumaParcial + k]$.Consideramos correcta esta implementación, ya que es análoga a enlistar los tickets en un vector, y en lugar de elegir el ticket de acuerdo a su número como en una lotería, se eligen de acuerdo a su posición en el vector (lo cual es equivalente, ya que los tickets son todos distintos y equiprobables).

La fórmula para obtener el ticket ganador es $= rand()\ \%cantidadTickets$, donde cantidadTickets es la cantidad total de tickets del sistema, $rand()$ es la función que genera una secuencia de números pseudoaleatorios del lenguaje C, alimentada por la semilla de parámetro y % es el operador resto. Esta fórmula genera una secuencia pseudoaleatorioa entre 0 y $cantidadTickets$, lo cual es suficiente para los experimentos a realizar.

La implementaciń del scheduler descripto hasta ahora se encuentra en los archivos **sched_lottery_base.cpp** y **sched_lottery_base.h**

2.4.8. Optimizaciones

Desde el punto de vista de performance, realizamos la optimización sugerida por los autores para agilizar la búsqueda de la tarea ganadora. Esta búsqueda se realiza en tiempo lineal sobre la lista de tareas, buscando a la tarea dueña del ticket ganador. Si ordenamos la lista según la cantidad de tickets de cada tarea de forma decreciente puede obtenerse una mejora de performance, ya que aquellas tareas con mayor cantidad de tickets tienen más probabilidad de poseer el ticket ganador. Dado que es necesario ordenar la lista, lo realizaremos sólo en caso de que cambie la distribución de tickets. En nuestra configuración del scheduler, esto va a suceder cuando una tarea se desbloquee y deba recibir tickets compensatorios, introducidos en la siguiente sección. Como esta diferencia de tickets se mantiene por sólo un tick, en muchos casos ordenar para luego realizar la búsqueda no genere una mejora, pero de todas formas decidimos agregar la optimización al scheduler para posibles distintas configuraciones de la distribución de tickets,

Desde el punto de vista algorítmico, los autores proponían diversas modificaciones que refinaban el modelo de *Lottery Scheduling*. El scheduler posee la optimización de **tickets compensatorios** previamente descripta.

Cuando se bloquea una tarea, nuestra implementación de los tickets compensatorios calcula la fracción f de quantum que ésta utilizó. Acto seguido multiplica la cantidad de tickets de la tarea por $1/f$. Como cada tarea posee la misma cantidad de tickets, ahora la tarea en cuestión tiene $1/f$ más probabilidad de salir que las demás, siendo efectivamente compensada en la elección del siguiente quantum a su desbloqueo. Por ejemplo, en un sistema con 100 tickets y 10 tareas, cada tarea posee un 10 % de probabilidad. En caso de que una tarea utilize 1/3 de su quantum, en elsiguiente quantum a ser desbloqueada se multiplicarán sus tickets por 1/(1/3), es decir tendra 30 tickets.

Calculando la probabilidad como $\#ticketsTarea/\#ticketsTotales$, vemos que la tarea compensada posee $30/120 = 25\,\%$ de probabilidad de ser elegida, mientras que las demás poseen $10/120 = 8{,}33\,\%$ de ser elegidas, siendo aproximadamente el triple de veces más probable por el quantum siguiente a su desbloqueo.

La implementación de tickets compensatorios obliga la introducción de las siguientes estructuras de datos:

- $map < tarea, par < multiplicador, bool >>$ aCompensar: almacena las tareas bloqueadas que deben ser compensadas cuando se produzca su desbloqueo, junto con el multiplicador de sus tickets y una variable bool que indica si ya se desbloqueo.

Y de las siguientes modificaciones a las funciones e introducción de una nueva:

- $tick()$: en caso de que una tarea se haya bloqueado sin consumir el total de su quantum, llama a la función $compensar()$.
- $compensar()$: agrega a la funcion que se acaba de bloquear al diccionario $aCompensar$, junto con su multiplicador correspondiente e indicando que aun no se desbloqueo.
- $unblock()$: si la tarea desbloqueda tiene una entrada en el diccionaro a $aCompensar$, significa que cuando se bloqueó no consumió el total de su quantum y la función $aCompensar$ guardó su multiplicador. En ese caso, se indica que la tarea fue desbloqueada y se deben recalcular los tickets.
- $redistribuirTickets()$: Pasa a recibir un modo de ejecucion, 0 o 1. El modo 0 indica que fue llamada desde un load o un block, por lo cual si hay tareas ya compensadas, deben seguir estandolo porque no llegaron a el tick de cambio de quantum. Por esto, realiza una distrubicion equitativa entre todos los tickets y recalcula las compensaciones de todas las tareas que necesitan ser compensadas usando sus multiplicadores, siempre y cuando ya hayan sido desbloqueadas. En cambio, el modo 1 indica que fue llamada desde un tick, por lo cual debe eliminar todas las compensaciones y redistribuir equitativamente.

Nota: es necesario aclarar que nos basta una variable bool para saber si alguna tarea fue compensada, ya que luego se redistribuirán todos los tickets equitativamente, y es indistinto cual de todas era. En caso de que esta no sea la distribución elegida, se soluciona simplemente con un vector de bool, indicando cuales estan compensadas.

Las demás optimizaciones no fueron consideradas por las siguientes razones:

- **Transferencias de tickets:** las tareas simuladas son independientes, y no se bloquean esperando una respuesta de otra tarea.
- **Inflación de tickets:** las tareas simuladas ejecutan instrucciones fijas, en ningún momento ejecutan alguna sección crítica en la cual requieran más tiempo de CPU.

3. Parte III: Evaluando los algoritmos de scheduling

3.1. Introducción

En esta sección se evaluan 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, realize *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:

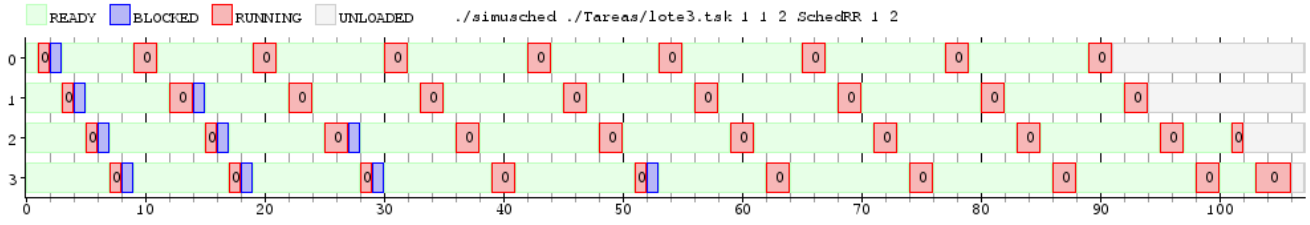
Turnaround: Es el intervalo de tiempo entre el momento en que el proceso comienza a ejecutarse por primera vez, hasta el momento en que el mismo termina. Es decir, es la suma de los períodos usados en esperar datos de memoria, en estar encolado en la "ready queue", ejecutandose en la CPU, y haciendo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

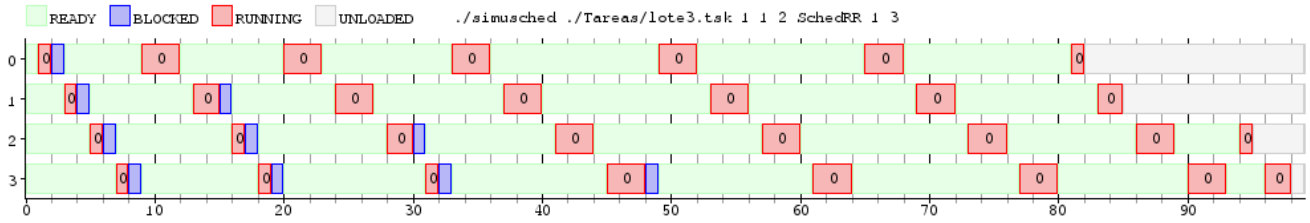
A continuación se pueden observar los resultados de la experimentación:



1 core, quantum = 2.

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

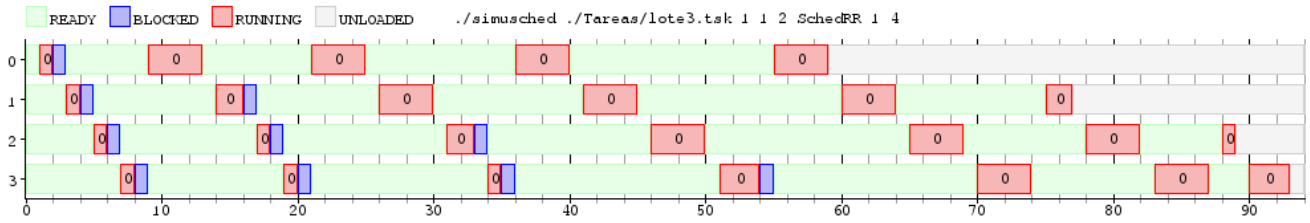
Waiting Time:
- P_0: 73
- P_1: 75
- P_2: 82
- P_3: 85
- Promedio: 78,75
- DS: 4,92



1 core, quantum = 3.

Turnaround:
- P_0: 81
- P_1: 82
- P_2: 90
- P_3: 91
- Promedio: 85,5
- DS: 4,55

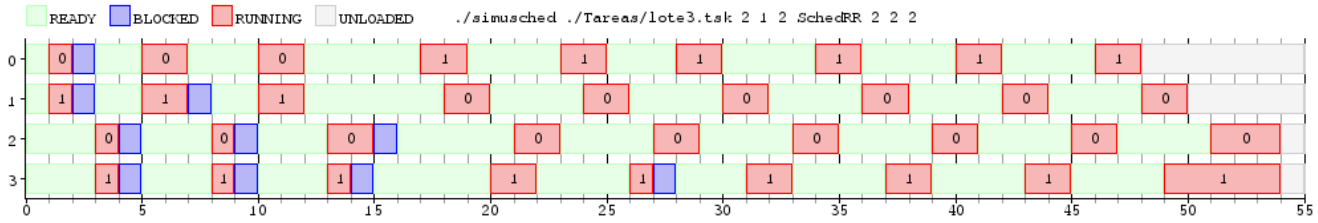
Waiting Time:
- P_0: 64
- P_1: 66
- P_2: 75
- P_3: 77
- Promedio: 70,5
- DS: 5,60



1 core, quantum = 4.

Turnaround:
- P_0: 58
- P_1: 74
- P_2: 84
- P_3: 86
- Promedio: 75,5
- DS: 10,33

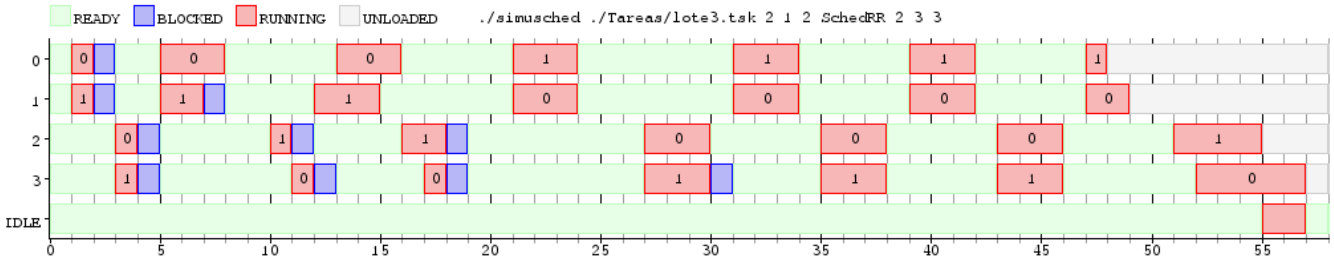
Waiting Time:
- P_0: 41
- P_1: 57
- P_2: 69
- P_3: 72
- Promedio: 59,75
- DS: 12,19



2 core, quantum = 2.

Turnaround:
- P_0: 47
- P_1: 49
- P_2: 51
- P_3: 51
- Promedio: 49,5
- DS: 1,66

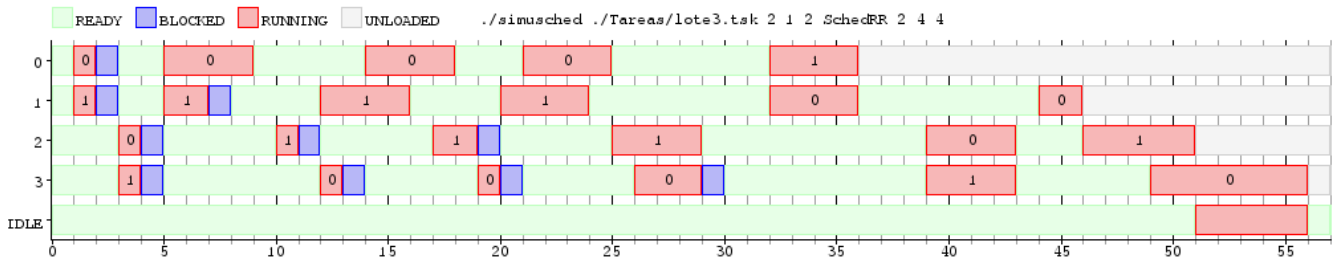
Waiting Time:
- P_0: 30
- P_1: 31
- P_2: 50
- P_3: 33
- Promedio: 36
- DS: 8,15



2 core, quantum = 3.

Turnaround:
- P_0: 47
- P_1: 48
- P_2: 52
- P_3: 54
- Promedio: 50,25
- DS: 2,86

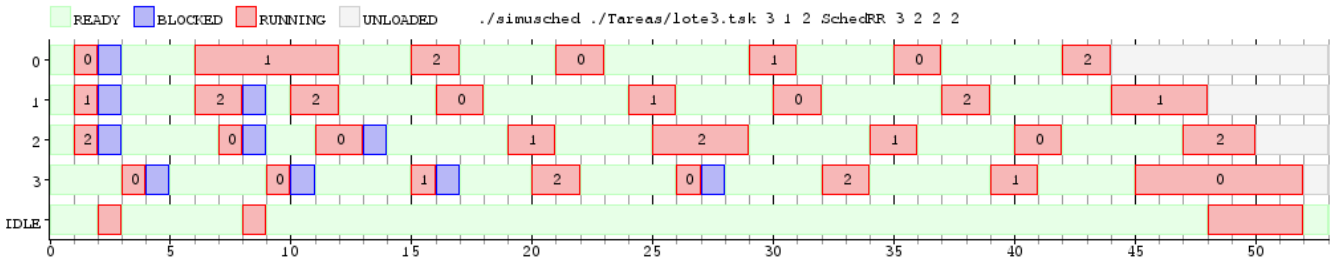
Waiting Time:
- P_0: 30
- P_1: 30
- P_2: 35
- P_3: 36
- Promedio: 32,75
- DS: 2,77



2 core, quantum = 4.

Turnaround:
- P_0: 35
- P_1: 45
- P_2: 48
- P_3: 53
- Promedio: 45,25
- DS: 6,57

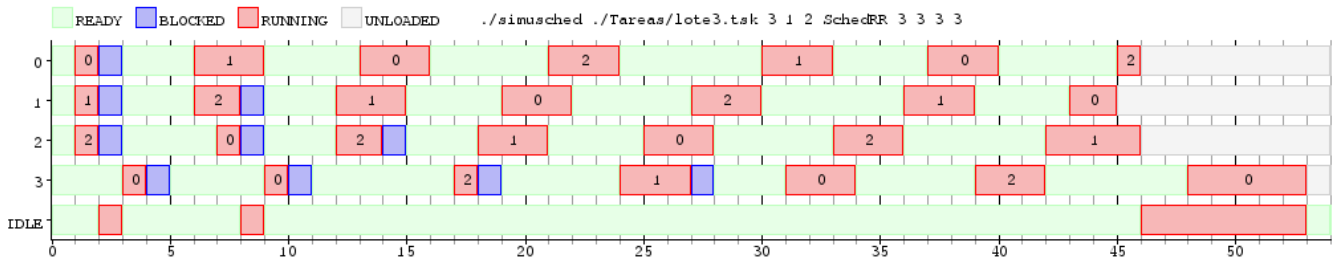
Waiting Time:
- P_0: 18
- P_1: 27
- P_2: 31
- P_3: 35
- Promedio: 27,75
- DS: 6,30



3 core, quantum = 2.

Turnaround:
- P_0: 43
- P_1: 47
- P_2: 49
- P_3: 49
- Promedio: 47
- DS: 2,45

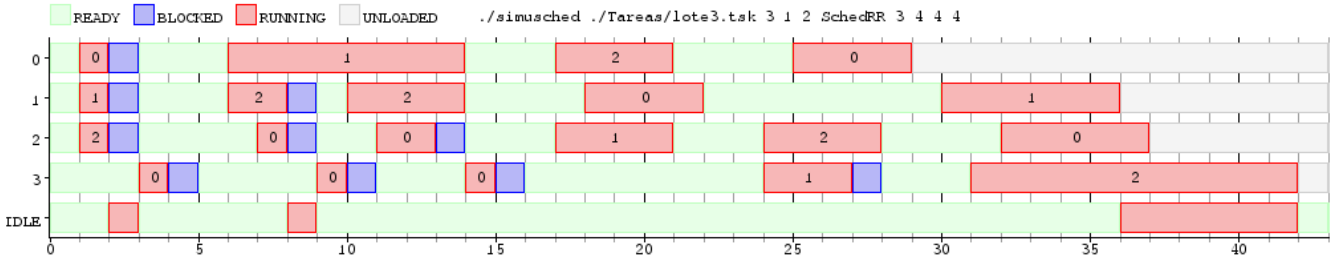
Waiting Time:
- P_0: 26
- P_1: 29
- P_2: 30
- P_3: 31
- Promedio: 29
- DS: 1,87



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

Las conclusiones que pudimos sacar fueron las siguientes:

- Dada una cantidad i de cores, a medida que se aumenta el quantum, el tiempo promedio de Turnaround y Waiting Time disminuye. Esto en principio indicaría que es buena idea aumentar el quantum, pero sin embargo a medida que se aumenta este, aumenta también el desvío estándar, lo cual implica que los valores de Turnaround y Waiting Time se encuentran más dispersos. Esto significa que va a haber procesos que terminen relativamente rápido mientras que otros no.
- Dado un mismo valor de quantum, aumentar la cantidad de cores siempre disminuye el tiempo promedio de Turnaround y Waiting Time, manteniendo o reduciendo a su vez el desvío estándar. Por lo tanto, obviamente, un aumento en la cantidad de cores siempre es beneficioso.

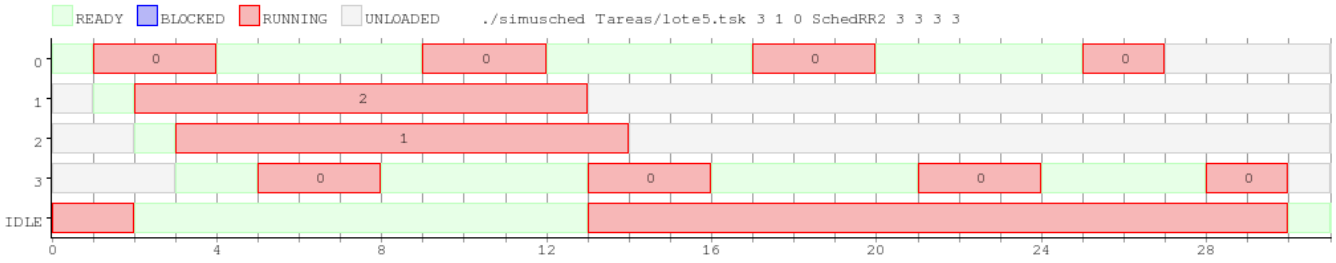
- El Waiting Time representa una parte muy importante del Turnaround: aproximadamente un 81 % en el caso de 1 core, 65 % con 2 cores, y 53 % con 3 cores.
- Lógicamente, los procesos que más llamadas bloqueantes realizan son los que más tiempo tardan en terminar. Sin embargo, en la mayor parte de los casos, la diferencia en los valores de Turnaround no es demasiado significativa.
- El valor óptimo del quantum para ambas métricas sería 4, si se tolera el desvío estándar asociado.

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.

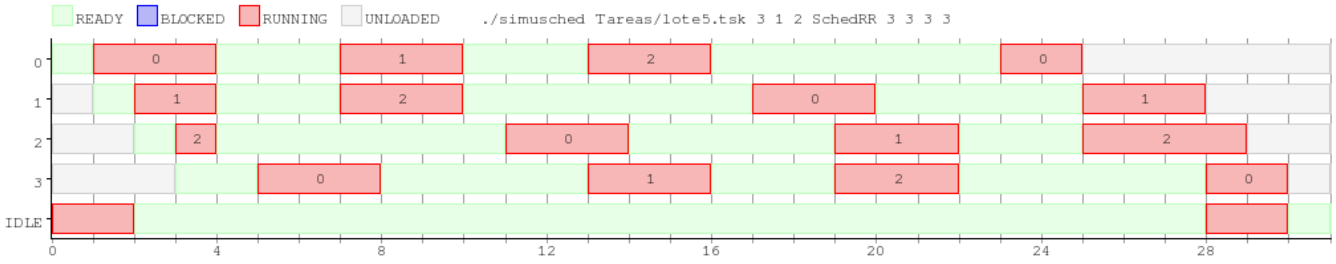
Cuando el proceso es bloqueado debemos tener una manera de saber a qué CPU pertenecía dicho proceso, y para tal fin cada core también tiene una cola de `BLOQUED`. Entonces, cuando un proceso es desbloqueado simplemente se lo busca en alguna de las listas `BLOQUED` de los cores, y una vez que se lo encuentra 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 estan igualmente cargados. Luego todas siguen ejecutándose en sus respectivos cores sin cambios de lugar como es esperado.



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

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

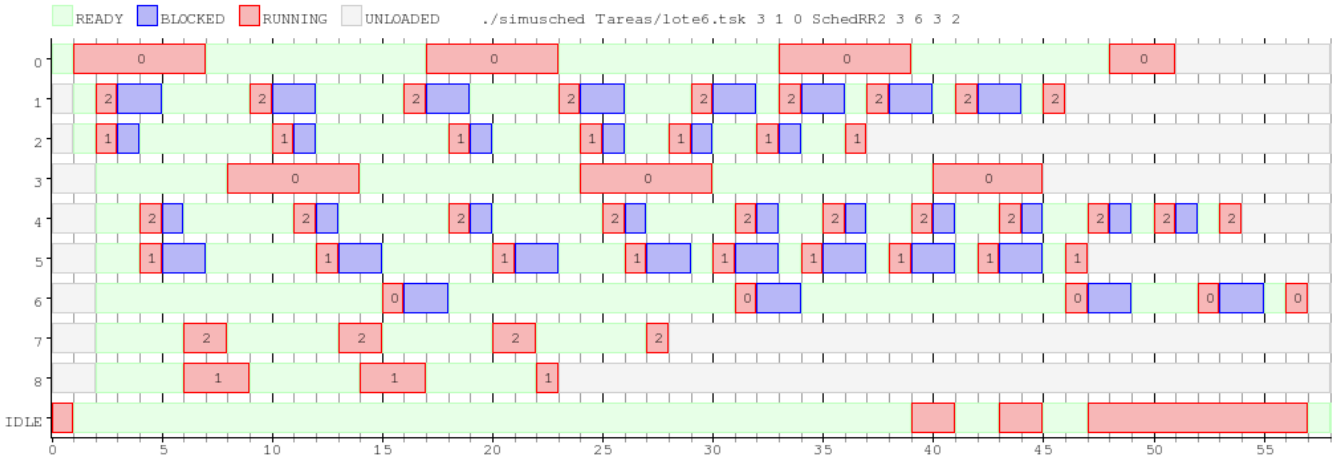
Para este lote de tareas en particular podemos apreciar que `SchedRR2` y `SchedRR` finalizan la ejecución de todas las tareas en un tiempo similar. Pero en el primer caso dos de las tareas finalizan rápido, algo que no sucede en el segundo caso, sin contar que en `SchedRR2` no hay costos por cambio de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

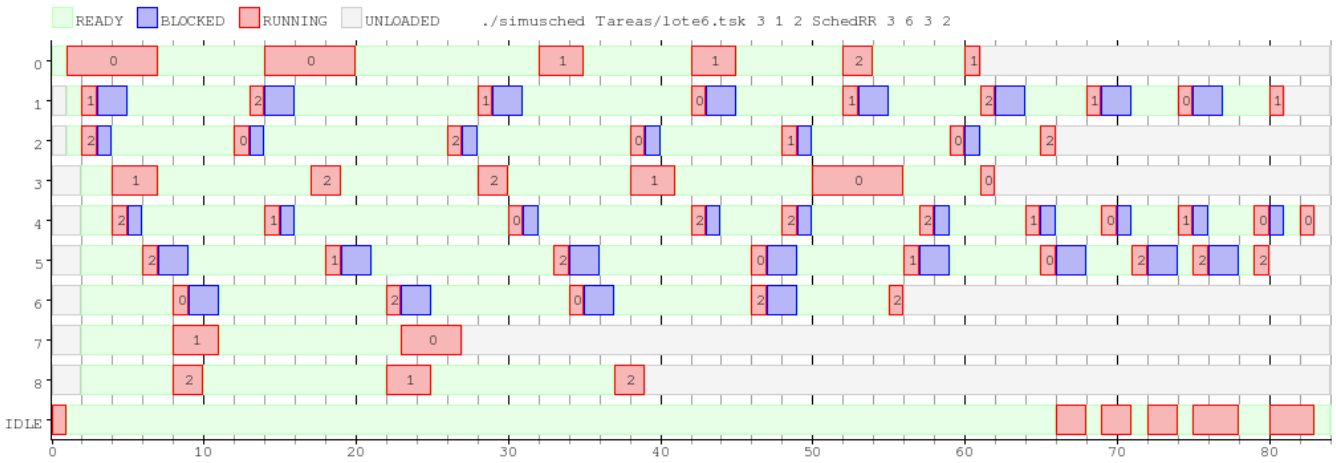
A continuación vamos a correr nuevamente los dos schedulers con el lote de tareas `lote6.tsk` que resultará en tareas pesadas para el core 0, y tareas mas livianas para el resto. El primer core tendrá un quantum mas grande que los otros. La idea es mostrar un escenario en donde puedo aprovechar cierta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

La idea detrás del diseño de este lote es construirlo para que el core 0 tome las tareas mas 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 mas alto que los demás. Así queda el procesamiento del `lote6.tsk` con `SchedRR2`:



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

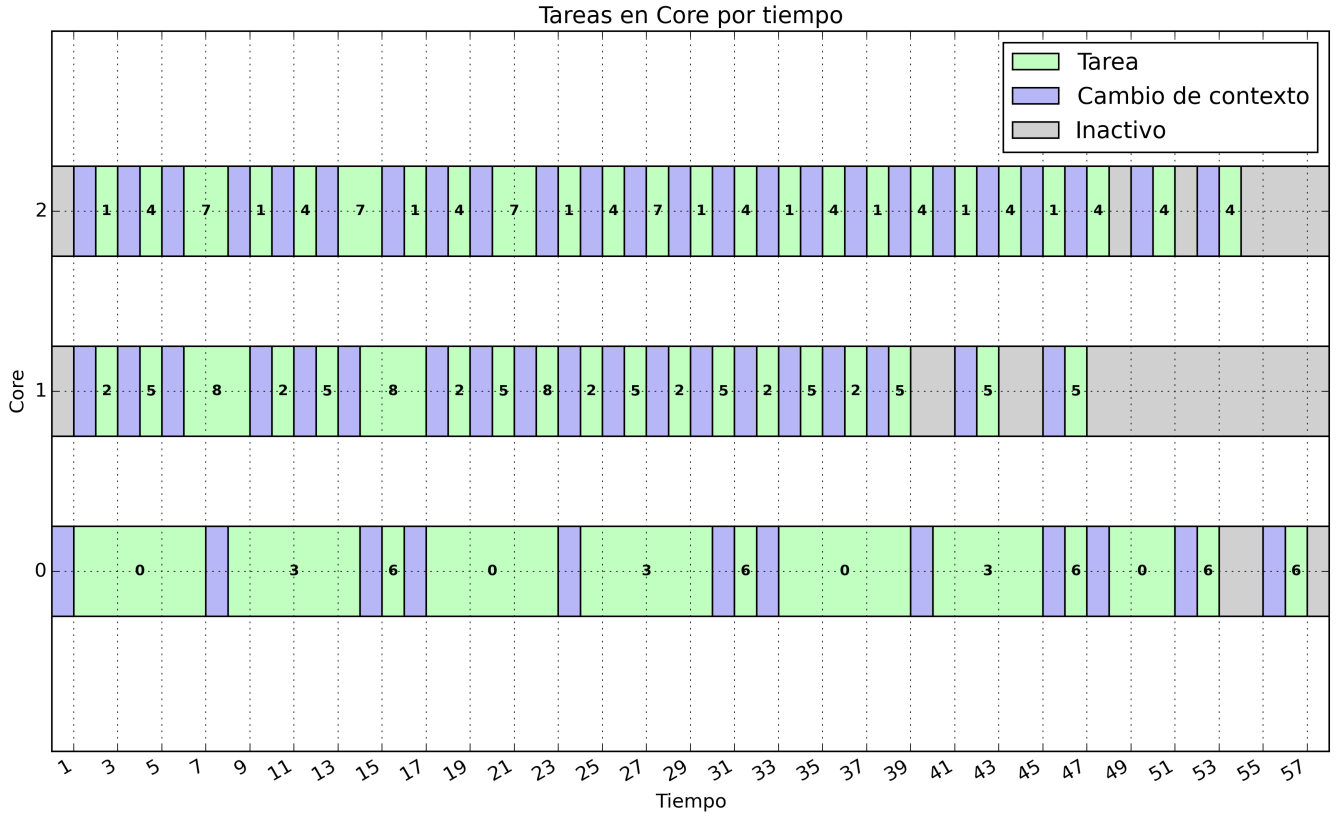
Y con `SchedRR` el procesamiento de dicho lote queda de la siguiente forma:



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

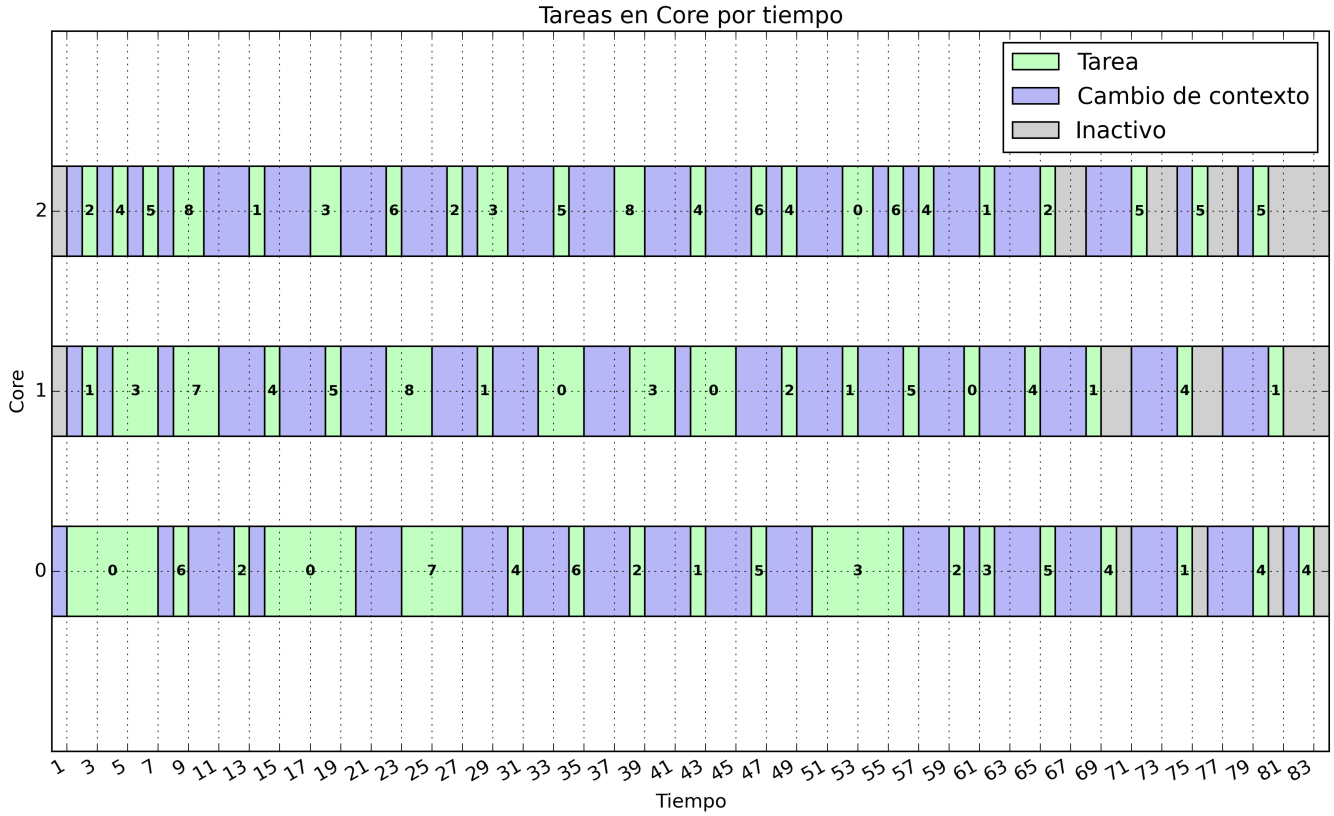
Vemos que para este lote, `SchedRR2` tiene mejor performance. Las tareas son finalizadas más rápido, y todo el lote es procesado más rápido que en `SchedRR`. Aún así en `SchedRR2` tenemos bastante tiempo de procesamiento en donde un procesador u otro permanece ocioso. Eso ocurre también en `SchedRR`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

Para ver esto más claro, a continuación mostramos los graficos de las actividades de los cores a lo largo del tiempo para el `lote6.tsk` usando exactamente los mismos parámetros. El primero es usando el scheduler `RR2`:



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

El siguiente es con el scheduler `RR`:



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

En este segundo gráfico podemos ver mejor cómo es que el tiempo de cambio de core afecta al tiempo total de procesamiento del lote. El costo de cambio de core es nulo en `RR2` debido a que las tareas se asignan a un core fijo hasta que finalizan. Algunas conclusiones que podemos sacar con esta corrida de lotes y las de puntos anteriores de `RR` son:

- El rendimiento de ambos schedulers depende mucho del contexto, es decir, de cómo vienen dadas las tareas, sus tipos, duraciones, orden, etc. Hay ciertos lotes que favorecen el uso de `RR` y hay otros que favorecen el uso de `RR2`.
- A nivel implementativo, `RR` es más fácil que `RR2`.
- Una de las ventajas de `RR2` es que no se paga el costo de cambio de cores, ya que cada tarea posee un único core en el que se ejecuta.
- Una desventaja de `RR2` es que puede, en algunas situaciones, dejar cores ociosos. `RR` también lo hace pero en mucha menor medida. En el caso de `RR` depende más que nada de la cantidad de tareas que quedan en relación al número de cores. En caso de `RR2`, pueden quedar muchas tareas asignadas a un solo core y ninguna al resto.
- Si hubiese a priori una política de balance de carga de los cores para `RR2`, estos estarían menos tiempo ociosos y mejoraría la performance del scheduler.

4. Ejercicio 9

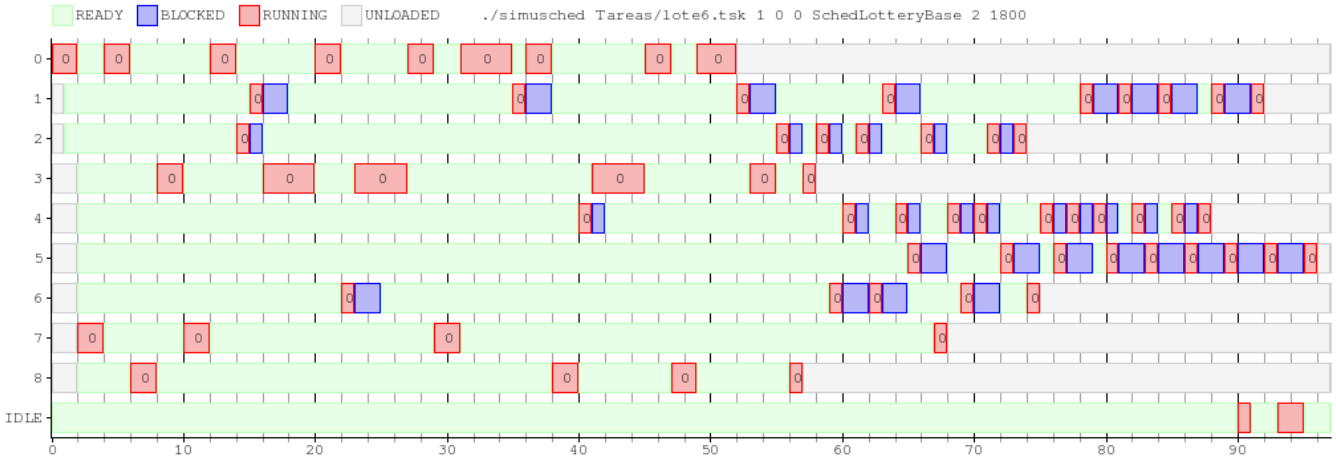
En esta parte del trabajo vamos a analizar la *fairness* del algoritmo de scheduling `SchedLottery`. Tomamos como definición de *fairness* al hecho de que cada proceso reciba igual cantidad de tiempo de CPU, o más precisamente, un tiempo apropiado para cada proceso de acuerdo a su prioridad y carga de trabajo.

A continuación vamos a mostrar los experimentos realizados para poder ver que efectivamente cuando se realizan n experimentos, el scheduler es realmente justo cuando n aumenta su tamaño. Notar que es necesario hacer más de una corrida ya que debido al factor pseudoaleatorio del algoritmo, realizar una o dos corridas no es suficiente para ver que efectivamente el scheduler es justo.

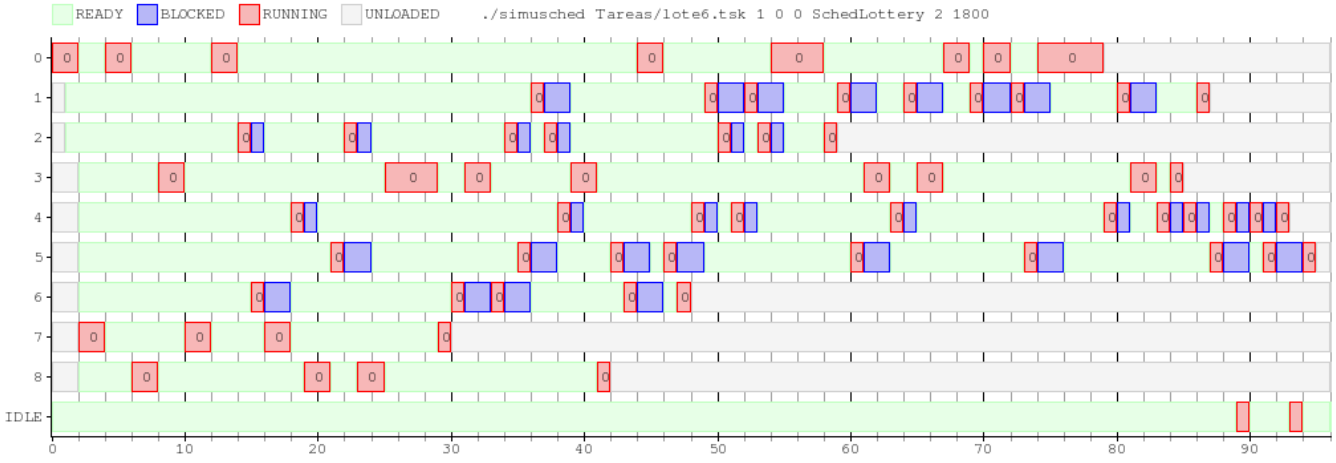
4.1. Ejercicio 10

Dadas las implementaciones de *ShedLottery*, con o sin tickets compensatorios (respectivamente en *sched$_l$ottery.cpp* y *sched$_l$otter$y_b$ase.cpp*), el ejercicio nos propone ponerlas a prueba y compararlas, relacionándolas con la problemática que presentaban los autores y verificar si los tickets compensatorios son una solución viable. A continuación mostramos los experimentos realizados:

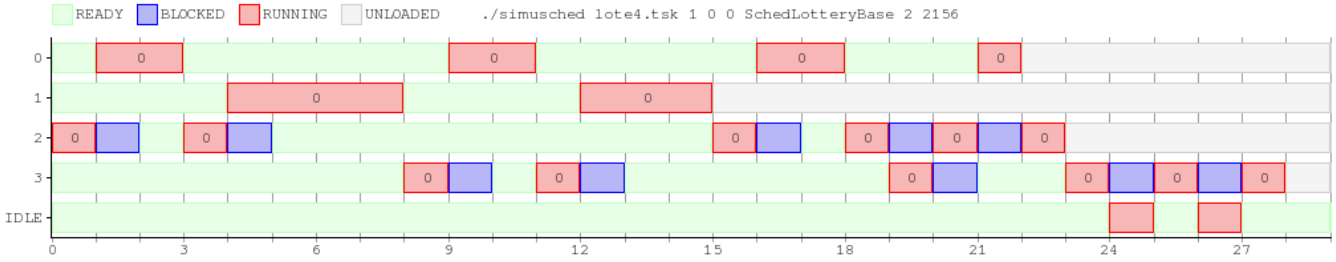
Para comenzar, realizamos experimentaciones sobre lotes ya utilizados en otros ejercicios. Para estos experimentos utilizamos semillas aleatorias ya que es indistinto, y Quantum = 2, de esa forma hay muchos cambios de tarea donde se realizan loterías.



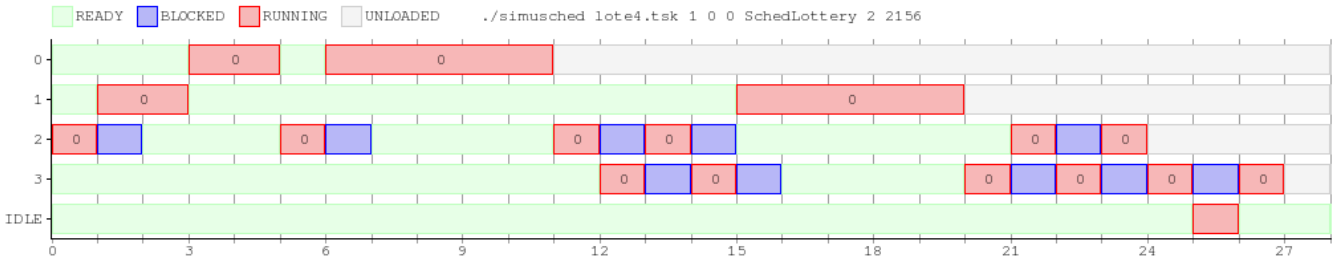
Lote = 6, Scheduler = Base, Quantum = 2, Semilla = 1800.



Lote = 6, Scheduler = Compensatorio, Quantum = 2, Semilla = 1800.

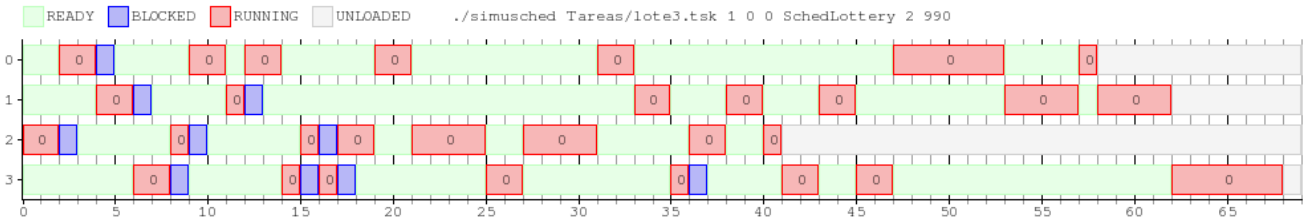


Lote = 4, Scheduler = Base, Quantum = 2, Semilla = 2156.



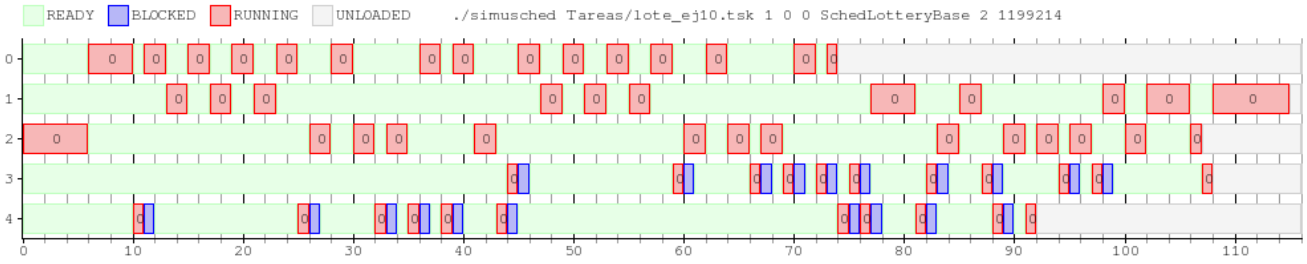
Lote = 4, Scheduler = Compensatorio, Quantum = 2, Semilla = 2156.

Lote = 3, Scheduler = Base, Quantum = 2, Semilla = 990.

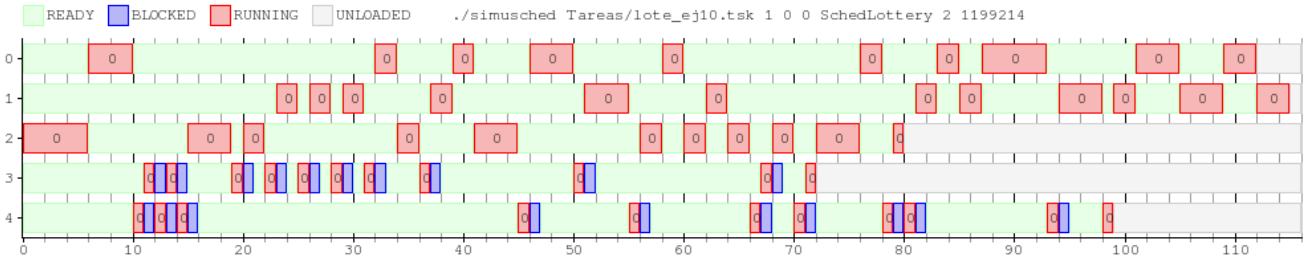


Lote = 3, Scheduler = Compensatorio, Quantum = 2, Semilla = 990.

Para este test generamos un lote de tareas nuevo, en el cual ejecutamos 3 tareas de uso intensivo de CPU por 30 ticks, y 2 tareas TaskConsola que realicen 10 llamadas bloqueantes de 1 tick de duración,



Lote = ej10, Scheduler = Base, Quantum = 2, Semilla = 1199214.



Lote = ej10, Scheduler = Compensatorio, Quantum = 2, Semilla = 1199214.

Puede apreciarse en todos los casos como en el scheduler base, las tareas que utilizan IO se amontonan en el final de la ejecución, bloqueándose y no retomando control del cpu hasta mucho después. En cambio, en el scheduler compensatorio pueden apreciarse algunas tareas en la mitad de la ejecución que luego de desbloquearse vuelven a tomar control de la CPU, gracias a la compensación.

5. Referencias

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

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