



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA



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

TP1 - Scheduling

Sistemas Operativos

Segundo Cuatrimestre de 2015

Apellido y Nombre	LU	E-mail
Shaurli, Tomas	671/10	zeratulzero@hotmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Contents

1	Introducción	3
2	Desarrollo y Resultados	4
3	Parte I – Entendiendo el simulador simusched	4
3.1	Ejercicios	4
3.2	Resultados y Conclusiones	4
3.2.1	Resolución Ejercicio 1	4
3.2.2	Resolución Ejercicio 2	5
3.2.3	Resolución Ejercicio 3	6
4	Parte II: Extendiendo el simulador con nuevos schedulers	8
4.1	Ejercicios	8
4.2	Resultados y Conclusiones	8
4.2.1	Resolución Ejercicio 4	8
4.2.2	Resolución Ejercicio 5	9
4.2.3	Resolución Ejercicio 5	12
4.2.4	Resolución Ejercicio 5	13
4.2.5	Resolución Ejercicio 8	14
5	Bibliografía	17

1 Introducción

En este Trabajo Práctico estudiaremos diversas implementaciones de algoritmos de scheduling. Haciendo uso de un simulador provisto por la cátedra podremos representar el comportamiento de estos algoritmos. Implementaremos dos Round-Robin, uno que permite migración de tareas entre núcleos y otro que no y a través de experimentación intentaremos comparar ambos algoritmos. Asimismo, basándonos en un scheduling el cual solo tenemos la versión ejecutable realizaremos una serie de experimentos para luego implementar el mismo.

2 Desarrollo y Resultados

3 Parte I – Entendiendo el simulador simusched

3.1 Ejercicios

- **Ejercicio 1** Programar un tipo de tarea `TaskConsola`, que simulara una tarea interactiva. La tarea debe realizar n llamadas bloqueantes, cada una de una duracion al azar 1 entre $bmin$ y $bmax$ (inclusive). La tarea debe recibir tres parametros: n , $bmin$ y $bmax$ (en ese orden) que seran interpretados como los tres elementos del vector de enteros que recibe la funcion. Explique la implementacion realizada y grafique un lote que utilice el nuevo tipo de tarea.
- **Ejercicio 2** Rolando, uno de los investigadores del departamento, utiliza su computadora para correr un algoritmo muy complejo que hace un uso intensivo de la CPU por 100 ciclos y no utiliza ninguna llamada bloqueante. Mientras corre su algoritmo suele poner su cancion preferida y luego navegar por internet (estas tareas realizan 20 y 25 llamadas bloqueantes respectivamente con una duracion variable entre 2 y 4 ciclos).
Escribir el lote de tareas que simule la situacion de Rolando. Ejecutar y graficar la simulacion usando el algoritmo FCFS para 1 y 2 nucleos con un cambio de contexto de 4 ciclos.
Calcular la latencia de cada tarea en los dos graficos. Explicar que desventaja tendria Rolando si debe mantener este algoritmo de scheduling y solo tiene disponible una computadora con un nucleo (haga referencia a los graficos y a los calculos anteriores para justificar su explicacion).
- **Ejercicio 3** Programar un tipo de tarea `TaskBatch` que reciba dos parametros: total cpu y cant bloqueos. Una tarea de este tipo debera realizar cant bloqueos llamadas bloqueantes, en momentos elegidos pseudoaleatoriamente. En cada tal ocasion, la tarea debera permanecer bloqueada durante exactamente un (1) ciclo de reloj.
El tiempo de CPU total que utilice una tarea `TaskBatch` debera ser de total cpu ciclos de reloj (incluyendo el tiempo utilizado para lanzar las llamadas bloqueantes; no asi el tiempo en que la tarea permanezca bloqueada). Explique la implementacion realizada y grafique un lote que utilice 3 tareas `TaskBatch` con parametros diferentes y que corra con el scheduler FCFS.

3.2 Resultados y Conclusiones

3.2.1 Ejercicio 1

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

Realizamos un ciclo de $i < \text{params}[0]$, donde utilizamos la función dada por la catedra, `uso_IO` a la cual le pasamos el pid correspondiente y un entero ciclos que es el valor random obtenido entre $bmin$ y $bmax$. Esa función `uso_IO` simula una llamada bloqueante.

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

A continuacion, el codigo mencionado:

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

Como experimentacion utilizamos trabajamos con el siguiente lote:

```
TaskConsola 5 3 7
TaskConsola 2 3 3
TaskConsola 15 2 7
```

De aqui, obtuvimos los siguientes resultados:

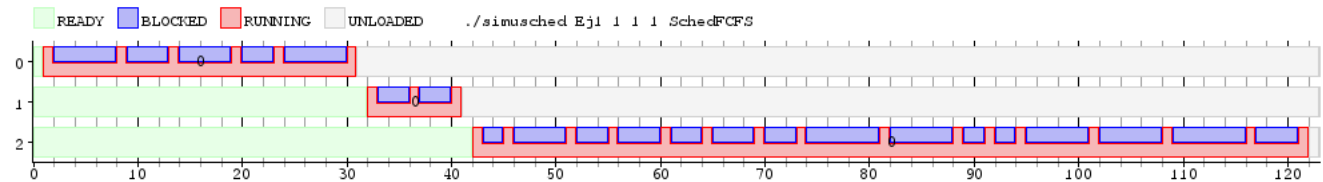


Grafico1 Scheduler FCFS - 1 core

3.2.2 Ejercicio 2

Para este punto, utilizamos el siguiente lote de tareas:

```
TaskCPU 100
TaskConsola 20 2 4
TaskConsola 25 2 4
```

El mismo, presenta una tarea de uso intensivo *TaskCPU* que dura 100 ticks, y otras dos interactivas, las cuales se bloquean 20 y 25 ticks respectivamente con una duración de entre 2 y 4 tanto para la primera como la segunda.

A continuación, los respectivos gráficos de mediciones.

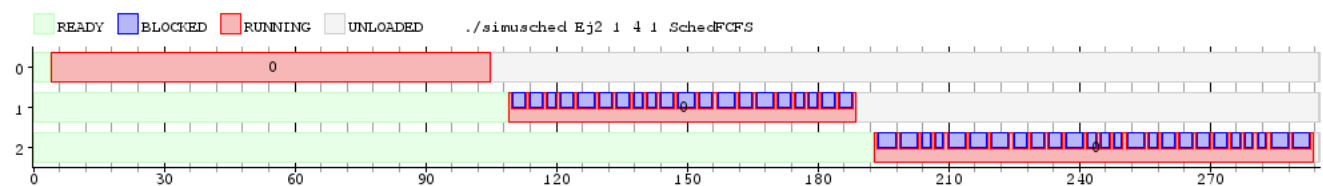


Grafico1 Scheduler FCFS - 1 core

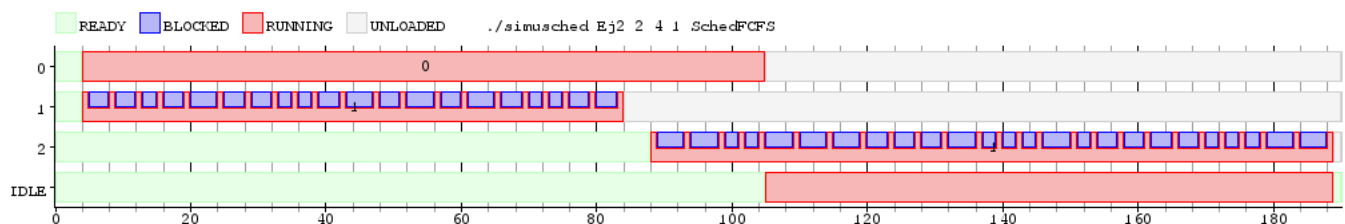


Grafico2 Scheduler FCFS - 2 core

A su vez, se nos solicito el calculo de la Latencia para cada una de las tareas basandonos en la experimentacion con uno y dos cores.

Latencia: Tiempo que demora una tarea en ejecutarse desde que la misma esta en estado *Ready*.

Para el grafico uno, los valores obtenidos para cada tarea fueron los siguientes:

- Tarea 1: 4
- Tarea 2: 109
- Tarea 3: 193

Para el grafico dos, los valores obtenidos para cada tarea fueron los siguientes:

- Tarea 1: 4
- Tarea 2: 4
- Tarea 3: 88

Se puede observar como aumenta el paralelismo a mayor cantidad de núcleos. En este scheduler en particular, esto ayuda de gran manera al rendimiento del sistema, puesto que un núcleo podrá ejecutar otra tarea recién cuando haya terminado la anterior. Las consecuencias de este comportamiento son visibles en los 2 graficos. Con un único core, solo se puede correr una tarea por vez, en este experimento comparado con usar 2 cores el tiempo se duplica.

3.2.3 Ejercicio 3

Al igual que con la tarea TaskConsola, mencionaremos nuestra implementación y por consiguiente explicaremos ciertos puntos de la misma.

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

Para este tipo de tarea, creamos un vector de tamaño igual a *total_cpu* el cual tendrá bool, ya sea true o false dependiendo del uso que se le de dentro de la tarea, ya sea uso_IO o uso_CPU. En caso de ser uso_IO sera true, y sino false.

Luego, utilizaremos un ciclo que irá desde 0 hasta el tamaño del vector y dependiendo el valor booleano, usará la funciones dadas por la catedra uso_IO o uso_CPU.

El experimento realizado para este nuevo tipo de tarea fue el siguiente:

Con un lote de tareas:

```
TaskBatch 10 3
TaskBatch 5 4
TaskBatch 8 1
```

Obtuvimos el siguiente diagrama:

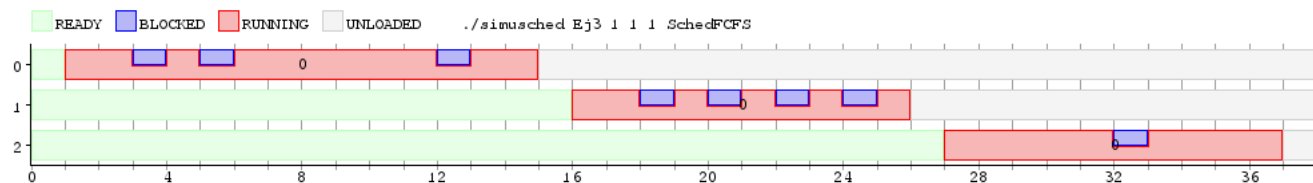


Grafico1 Scheduler FCFS - 1 core

Conclusiones

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

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

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

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

4 Parte II: Extendiendo el simulador con nuevos schedulers

4.1 Ejercicios

- **Ejercicio 4** Completar la implementación del scheduler Round-Robin implementando los metodos de la clase SchedRR en los archivos sched_rr.cpp y sched_rr.h. La implementacion recibe como primer parametro la cantidad de nucleos y a continuacion los valores de sus respectivos quantums. Debe utilizar una unica cola global, permitiendo asi la migracion de procesos entre nucleos.
- **Ejercicio 5** Diseñe un lote con 3 tareas de tipo TaskCPU de 50 ciclos y 2 de tipo TaskConsola con 5 llamadas bloqueantes de 3 ciclos de duracion cada una. Ejecutar y graficar la simulacion utilizando el scheduler Round-Robin con quantum 2, 10 y 50.
Con un cambio de contexto de 2 ciclos y un solo nucleo calcular la latencia, el waiting time y el tiempo total de ejecucion de las cinco tareas para cada quantum. ¿En cual es mejor cada uno? ¿Por que ocurre esto?
- **Ejercicio 6** Grafique el mismo lote de tareas del ejercicio anterior para el scheduler FCFS. Haciendo referencia a lo que se observa en los graficos de este ejercicio y el anterior, explique las diferencias entre un scheduler Round-Robin y un FCFS.
- **Ejercicio 7** El scheduler SchedMystery fue creado por docentes investigadores de nuestra materia y ha sido destacado en la ultima publicacion de ACM - SIGOPS, Operating Systems Review. Desde entonces, numerosos investigadores de todo el mundo nos han contactado para pedirnos su codigo fuente. Sin embargo, su codigo no aparece en ninguno de los repositorios de la materia y nadie parece recordar quienes habian estado detras de su implementacion. Se les pide experimentar con dicho scheduler (aprovechando que hemos conseguido el codigo objeto) y replicar su funcionamiento en SchedNoMystery. Graficar como maximo tres lotes de tareas utilizados en los experimentos y explicar en cada uno por separado que caracteristicas de SchedMystery identificaron con ese lote. Nota: El scheduler funciona para un solo core y toma uno o mas argumentos numericos.
- **Ejercicio 8** Implemente un scheduler Round-Robin que no permita la migracion de procesos entre nucleos (SchedRR2). La asignación de CPU se debe realizar en el momento en que se produce la carga de un proceso (load). El nucleo correspondiente a un nuevo proceso sera aquel con menor cantidad de procesos activos totales (RUNNING + BLOCKED + READY). Explique un escenario real donde la migracion de nucleos sea beneficiosa y uno donde no (mencione especificamente que metricas de comparacion vistas en la materia mejorarian en cada caso). Diseñe un lote de tareas en nuestro simulador que represente a cada uno de esos escenarios y grafique su resultado para cada implementacion. Calcule y compare en cada grafico las metricas que mencione.

4.2 Resultados y Conclusiones

4.2.1 Ejercicio 4

Para desarrollar la implementación del scheduler *Round – Robin* y que este funcione de una forma correcta utilizamos una serie de estructuras puntuales.

Las mismas son las siguientes:

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

5. Una lista de *bloqueados* esta tendra procesos que se bloquearon cuando estaban corriendo.

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

A su vez, tomamos ciertas decisiones en esta implementación:

- Si una tarea se encuentra bloqueada cuando se produce el tick del reloj, esta misma es desalojada de la cola global, y agregada en una lista de bloqueados. Además, sera reseteado el quantum, se le dará inicio a la próxima tarea que se encuentre ready y cuando el sistema operativo, nos envíe una señal de unblock, la tarea desalojada regresará al final de la cola global.

4.2.2 Ejercicio 5

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

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

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

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

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

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

Para corroborar que el comportamiento era el deseado, nos solicitaron 1 lotes de tareas compuestos por tareas del tipo *taskConsola* y *taskCpu*, trabajando con 1 cores y utilizando distintos *quantum* para cada uno de los mismos.

El lote de tareas fue el siguiente:

```
*3 TaskCPU 50
*2 TaskConsola 5 3 3
```

Obteniendo los siguientes resultados:



Lote1 - Scheduler RR - 1 core - 2 quantum

Latencia:

- Tarea 0: 2
- Tarea 1: 6
- Tarea 2: 10
- Tarea 3: 14

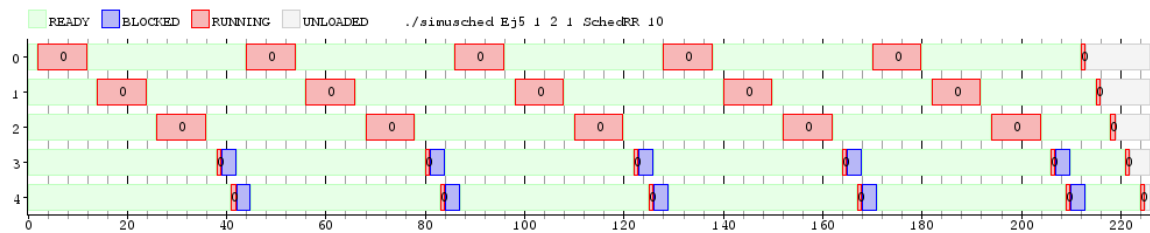
- Tarea 4: 18
- PROMEDIO: 10

Waiting time:

- Tarea 0: 289
- Tarea 1: 292
- Tarea 2: 295
- Tarea 3: 84
- Tarea 4: 87
- PROMEDIO: 209,4

Tiempo total de ejecución:

- Tarea 0: 339
- Tarea 1: 342
- Tarea 2: 345
- Tarea 3: 105
- Tarea 4: 108
- PROMEDIO: 247,8



Lote1 - Scheduler RR - 1 core - 10 quantum

Latencia:

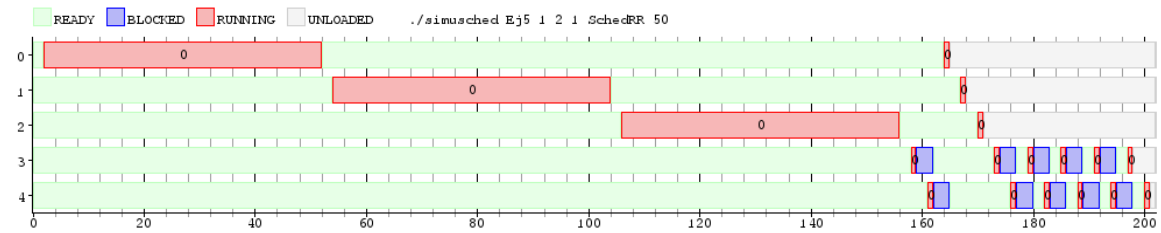
- Tarea 0: 2
- Tarea 1: 14
- Tarea 2: 26
- Tarea 3: 38
- Tarea 4: 41
- PROMEDIO: 24,2

Waiting time:

- Tarea 0: 162
- Tarea 1: 165
- Tarea 2: 198
- Tarea 3: 201
- Tarea 4: 204
- PROMEDIO: 186

Tiempo total de ejecución:

- Tarea 0: 213
- Tarea 1: 216
- Tarea 2: 219
- Tarea 3: 222
- Tarea 4: 225
- PROMEDIO: 219



Lote1 - Scheduler RR - 1 core - 50 quantum

Latencia:

- Tarea 0: 2
- Tarea 1: 54
- Tarea 2: 106
- Tarea 3: 158
- Tarea 4: 161
- PROMEDIO: 96,2

Waiting time:

- Tarea 0: 114
- Tarea 1: 117
- Tarea 2: 120
- Tarea 3: 177
- Tarea 4: 180
- PROMEDIO: 141,6

Tiempo total de ejecución:

- Tarea 0: 165
- Tarea 1: 168
- Tarea 2: 171
- Tarea 3: 198
- Tarea 4: 201
- PROMEDIO: 180,6

- El caso con mejor latencia es el primer experimento, esto se debe al escaso quantum que se le otorga a cada tarea permitiéndoles ejecutar por primera vez rápidamente.
- El 3er experimento tiene el mejor waiting time, esto se debe a su alto quantum. Debido a que el quantum es muy elevado se desperdicia la menor cantidad de ticks de reloj en cambio de contexto.
- Igual que en el caso anterior como el 3er experimento es el que menos ticks pierde por cambio de contexto debido al alto quantum, esta es la que menos tiempo de ejecución tiene.

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

Luego de estos experimentos pudimos observar ciertos puntos del comportamiento del Round-Robin:

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

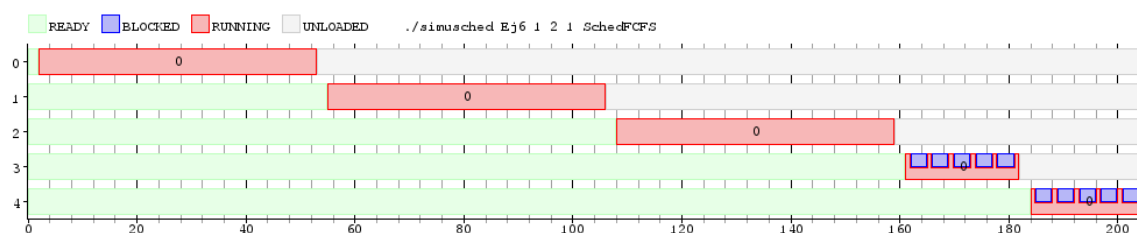
Finalmente, dado su carácter circular y equitativo, podemos afirmar que todas las tareas que estén en condiciones de correr serán ejecutadas y ninguna será negada de tiempo de procesamiento.

4.2.3 Ejercicio 6

El lote de tareas solicitado fue el siguiente:

```
*3 TaskCPU 50
*2 TaskConsola 5 3 3
```

Obteniendo los siguientes resultados:



Lote1 - Scheduler FCFS - 1 core

A modo de analisis, hemos obtenido las siguientes conclusiones:

- Un scheduler FCFS corre una tarea hasta que esta termina a diferencia de uno Round-Robin que las va intercambiando otorgándole un tiempo equitativo a cada tarea.
- Para que Round-Robin sea considerablemente mas eficiente que FCFS el quantum debe ser lo suficientemente alto para no tener altos valores de waiting-time durante los cambios de contexto pero no demasiado alto sino se comporta igual que un scheduler FCFS.
- Si se da el caso anterior el scheduler Round-Robin es eficiente en términos de latencia en comparación con FCFS.

4.2.4 Ejercicio 7

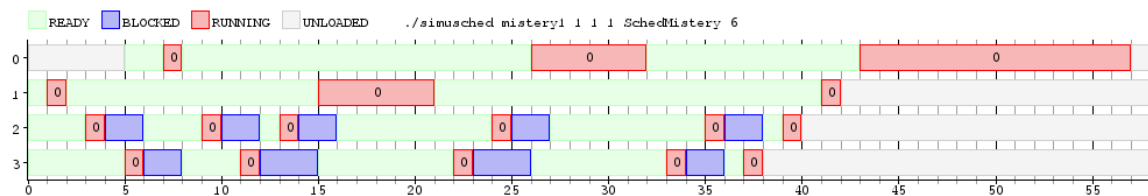
En este punto se nos solicitó experimentar con el código objeto de un SchedMystery y a partir de los mismos, realizar una réplica del mismo.

A continuación, expondremos tres experimentos de los cuales sacamos ciertas particularidades que presenta este Scheduler.

Con un lote de tareas compuesto por:

```
@5:
TaskCPU 20
@0:
TaskCPU 7
TaskConsola 5 2 2
@:8
TaskConsola 4 2 3
```

Obteniendo lo siguiente:



Lot1 - Sched Mystery - 1 core - Quantum = 6

De aquí, pudimos abstraer que, una tarea al bloquearse es desalojada y cuando la misma se desbloquea, el scheduler, le da una prioridad otorgándole $\text{quantum} = 1$ para que vuelva a correr y luego volver a encolarla en un orden cclico similar al de Round-Robin.

Con el siguiente lote:

```
TaskCPU 20
TaskCPU 10
@20:
TaskCPU 15
```

Obteniendo lo siguiente:



Lot2 - Sched Mystery - 1 core - Quantum = 5 3 2

De aquí, pudimos observar, la existencia de una prioridad a tareas *mas nuevas*, ya que al llegar la misma y estar ready, es la proxima a correr por el scheduler, y con un quantum igual a la sumatoria de los quantum ya pasados hasta el momento de ser cargada.

Se puede apreciar también un orden ciclico del mismo, y luego de dar esta prioridad mencionada por única vez, dicha tarea es encolada con un orden cíclico respetando el mismo quantum que el resto de las tareas.

Además, se puede observar distintos valores de quantums, inicialmente todos con 1, y luego, por cada vuelta cíclica los mismos van cambiando dependiendo del valor recibido como parámetro.(En nuestro

experimento, por ej, los valores fueron 5 3 y 2)

En conclusion:

- Se puede observar un caracter cíclico con posibilidad de varios quantums distintos, los cuales van cambiando acorde a las vueltas cíclicas que da el scheduler.
- Prioridad para tareas bloqueantes (al ser desbloqueada es la primera en cargar otorgándole quantum = 1)
- Prioridad para tareas mas recientes (al llegar la misma y estar ready esta pasa a ser la próxima en correr con un quantum igual a la sumatoria de los quantums ya transcurridos hasta el momento)

4.2.5 Ejercicio 8

La idea principal de esta nueva versión de *Round – Robin* se centraliza en que no permita migración entre cores, esto se basa principalmente en utilizar una cola para cada núcleo por separado, y en cada cola respectiva se encolarán las tareas que fueron asignadas inicialmente a cada nucleo.

Para desarrollar este tipo de algoritmo, el cual denominaremos *RR2*, utilizamos estructuras puntuales, enunciadas a continuación:

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

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

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

Finalmente, cuando una tarea finaliza, la quitamos y descontamos una unidad a la posición *i* del vector *cantidad*. Esta es la única vez, en la cual se descuenta. Aunque una tarea se bloquee, la misma seguirá contando en el vector. De esta forma se cumplirá, que las tareas son asignadas a los cores con menor cantidad de tareas.

Luego de realizar dicha implementación, en comparación al Round-Robin original, hemos conjeturado las siguientes hipótesis:

1. Comportamiento menos eficiente en el RR2 con respecto al paralelismo, ya que al no permitir migración de nucleos este se pierde.
2. Comportamiento más eficiente en el RR2 con lotes de tareas que se bloquean un gran número de veces. Esto surge ya que el Round-Robin original, es mas proclive a realizar cambios de contexto con la posibilidad de darse un cambio de core.

Procedemos a demostrar la primer conjetura:

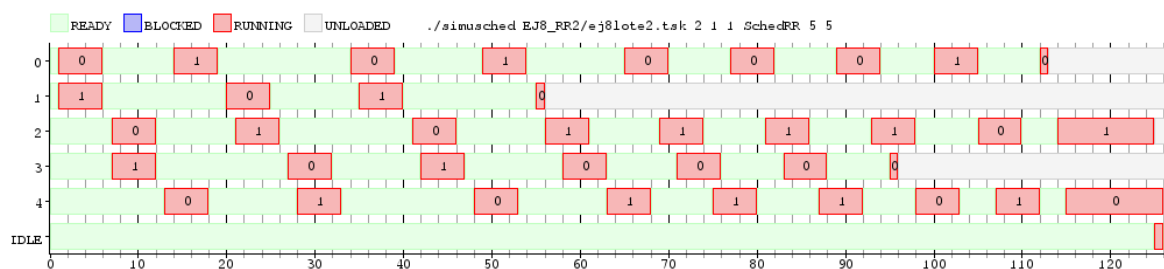
Comportamiento menos eficiente en el RR2 con respecto al paralelismo, ya que al no permitir migración de núcleos este lo pierde

Un ejemplo de esto, en la vida real seria, estar corriendo testeos de software y al mismo tiempo analizando algoritmos de alta complejidad para demostraciones matemáticas.

Un ejemplo de los lotes utilizados que demuestra esto fue el siguiente:

TaskCPU 40
TaskCPU 15
TaskCPU 50
TaskCPU 30
TaskCPU 50

Obteniendo los siguientes datos relevantes:



Lote3 - Round Robin - 2 core - Quantum = 5 - cambio de contexto = 1



Lote3 - Round Robin 2 - 2 core - Quantum = 5 - cambio de contexto = 1

Se puede ver en estos diagramas como la implementación del Round Robin original trabaja mejor finalizando la ejecución de las tareas hasta 50 milisegundos antes.

Esto se da por la falta de paralelismo de el RR2 ya que al ser asignados los procesos a cada core, cuando uno de los dos finaliza, este queda ocioso ya que no existe la posibilidad de migrar procesos.

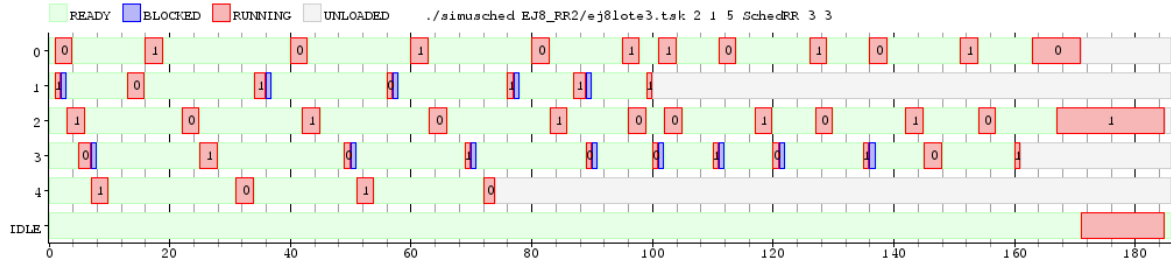
Luego, el RR2 resulta beneficioso:

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

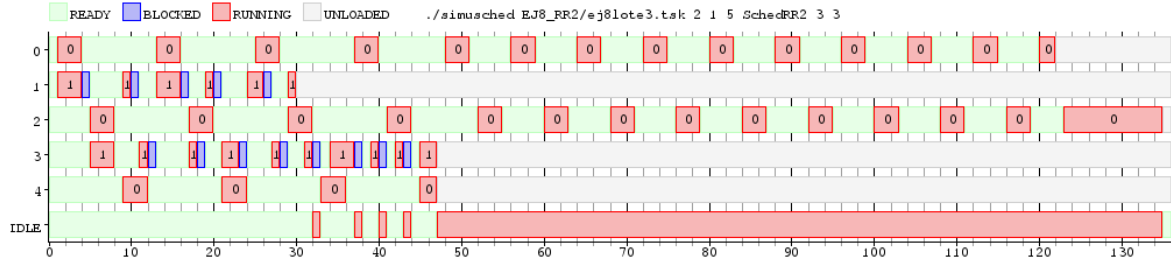
Un ejemplo de esto, puede ser al estar corriendo un juego (en el cual siempre se esta interactuando con el teclado) y además estar corriendo algun software.

Para esto,un lotes de tareas que ejemplifica lo dicho puede ser el siguiente:

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



Lote3 - Round Robin - 2 core - Quantum = 3 - cambio de contexto = 1



Lote3 - Round Robin 2 - 2 core - Quantum = 3 - cambio de contexto = 1

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

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

5 Bibliografía

- Cátedra de Sistemas Operativos - Clases teóricas y prácticas (2º Cuatrimestre 2015)
- Facultad de Ingenieria Uruguay
(https://eva.fing.edu.uy/pluginfile.php/75120/mod_resource/content/1/6-SO-Teo-Planificacion.pdf)
- Operating Systems Concepts, Abraham Silberschatz & Peter B. Galvin