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

TP1 - Scheduling

Sistemas Operativos

Primer Cuatrimestre de 2016

Apellido y Nombre	LU	E-mail
Tripodi, Guido	843/10	guido.tripodi@hotmail.com
Sueiro, Diego	75/90	dsueiro@gmail.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 7

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 6 12
4.2.4 Resolución Ejercicio 7 13
4.2.5 Resolución Ejercicio 8 15

5 Aclaraciones y Bibliografia **18**
5.1 Makefile 18
5.2 Bibliografia 18

1 Introducción

En este Trabajo Práctico estudiaremos diversas implementaciones de algoritmos de scheduling. Haciendo uso de un simulador provisto por la cátedra podremos reprensentar el comportamiento de estos algoritmos. Implementaremos dos 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 simulará una tarea interactiva. La tarea debe realizar n llamadas bloqueantes, cada una de una duración al azar 1 entre bmin y bmax (inclusive). La tarea debe recibir tres parámetros: n, bmin y bmax (en ese orden) que serán interpretados como los tres elementos del vector de enteros que recibe la función. Explique la implementación realizada y grafique un lote que utilice el nuevo tipo de tarea.
- **Ejercicio 2** El grupo de competencia de Data Mining, reciente ganador de importante concurso internacional, esta preparando el algoritmo para su próxima victoria. Para esto necesita utilizar fuertemente la CPU por 500 ciclos. A su vez, el grupo usa la máquina como servidor remoto, utilizando 3 usuarios que realizan llamadas bloqueantes de 10, 20 y 30 respectivamente y de una duración al azar de hasta 4 ciclos. Escribir el lote de tareas que simule la situación del grupo. Ejecutar y graficar la simulación usando el algoritmo FCFS para 1 y 2 y 4 núcleos con un cambio de contexto de 5 ciclos. Calcular la latencia de cada tarea en los dos gráficos. Explicar que desventaja tendría si debe mantener este algoritmo de scheduling y solo tiene disponible una computadora con un núcleo (haga referencia a los gráficos y a los cálculos anteriores para justificar su explicación).
- **Ejercicio 3** Programar un tipo de tarea TaskBatch que reciba dos parámetros: total cpu y cant bloqueos. Una tarea de este tipo debera realizar cant bloqueos llamadas bloqueantes, en momentos elegidos pseudoaleatoriamente. En cada tal ocasión, la tarea deberá permanecer bloqueada durante exactamente dos (4) ciclos de reloj.
  El tiempo de CPU total que utilice una tarea TaskBatch deberá ser de total cpu ciclos de reloj (incluyendo el tiempo utilizado para lanzar las llamadas bloqueantes; no así el tiempo en que la tarea permanezca bloqueada). Explique la implementación realizada y grafique un lote que utilice 4 tareas TaskBatch con parámetros 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 < 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 continuación, el código 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 experimentación trabajamos con el siguiente lote:

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

De aquí, obtuvimos los siguientes resultados:

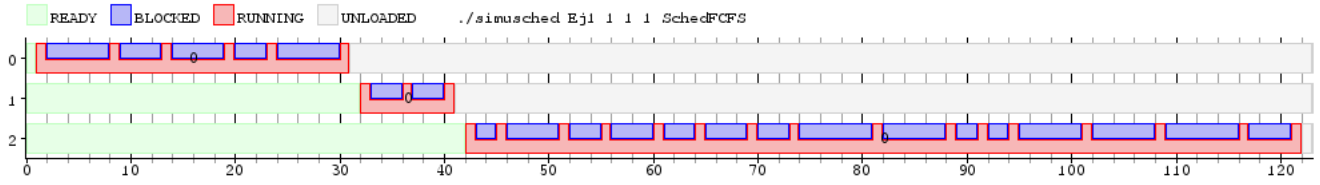


*Gráfico*1.1 Scheduler FCFS - 1 core

3.2.2 Ejercicio 2

Para este punto, utilizamos el siguiente lote de tareas:

```
TaskCPU 500
TaskConsola 10 2 4
TaskConsola 20 2 4
TaskConsola 30 2 4
```

El mismo, presenta una tarea de uso intensivo *TaskCPU* que dura 500 ticks, y otras tres interactivas, las cuales se bloquean 10, 20 y 30 veces respectivamente con una duración de entre 2 y 4 tanto para la primera, la segunda como la tercera.
A continuación, los respectivos gráficos de mediciones.

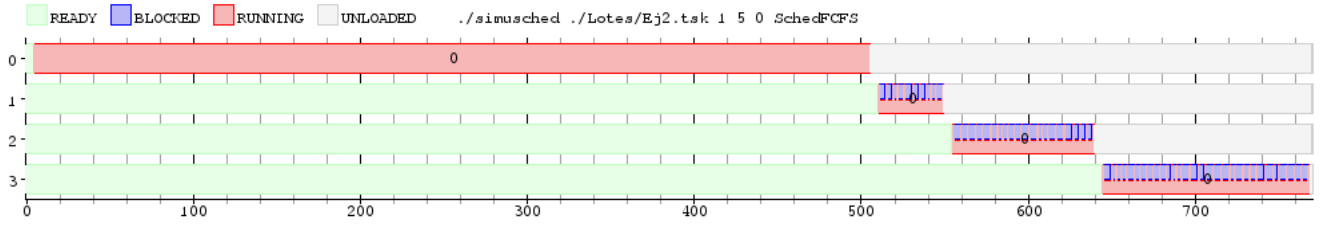


*Gráfico*2.1 Scheduler FCFS - 1 core

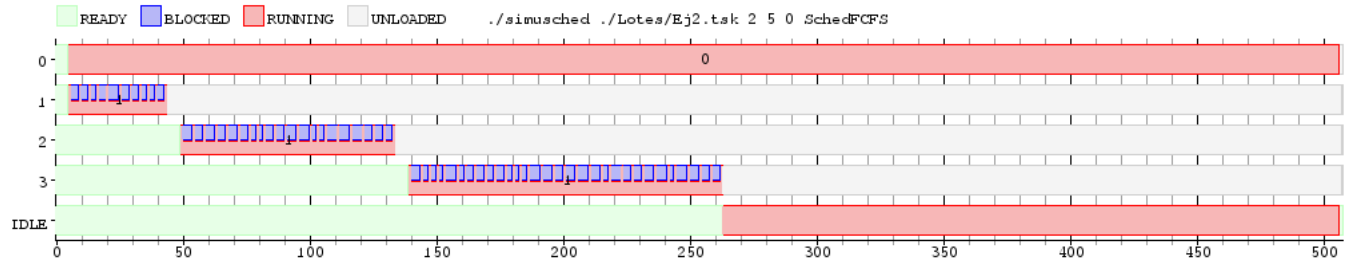


*Gráfico*2.2 Scheduler FCFS - 2 core

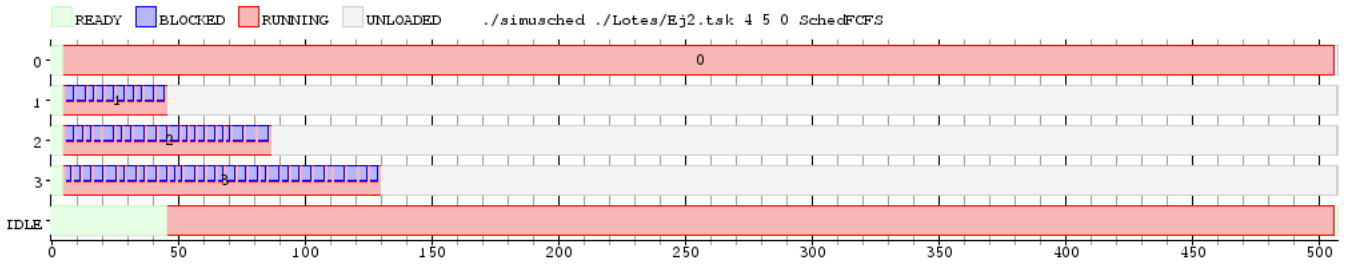


*Gráfico*2.3 Scheduler FCFS - 4 core

A su vez, se nos solicitó el cálculo de la Latencia para cada una de las tareas basándonos en la experimentación con uno, dos y cuatro cores.

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

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

- Tarea 0: 4
- Tarea 1: 510
- Tarea 2: 554
- Tarea 3: 644

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

- Tarea 0: 4
- Tarea 1: 4
- Tarea 2: 49
- Tarea 3: 139

Para el gráfico tres, los valores obtenidos para cada tarea fueron los siguientes:

- Tarea 0: 4
- Tarea 1: 4
- Tarea 2: 4
- Tarea 3: 4

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. Si el equipo continuara usando el scheduler FCFS no podría ejecutar otra tarea hasta que terminara de correr la anterior. Las consecuencias de este comportamiento son visibles en los 3 gráficos. 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 nuestro implementación y por consiguiente explicaremos ciertos puntos de la misma.

```
void TaskBatch(int pid, vector<int> params) {
     int total_cpu = params[0];
     int cant_bloqueos = params[1];
     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,2);
           else
              uso_CPU(pid, 1);
        }
}
```

Para este tipo de tarea, creamos un vector de tamaño igual a *total_cpu* el cual contiene valores booleanos, si el valor en el índice del vector es true este corresponderá a la función uso_IO, caso contrario uso_CPU.
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 3 3
TaskBatch 5 4
TaskBatch 4 4
TaskBatch 5 3
```

Obtuvimos el siguiente diagrama:

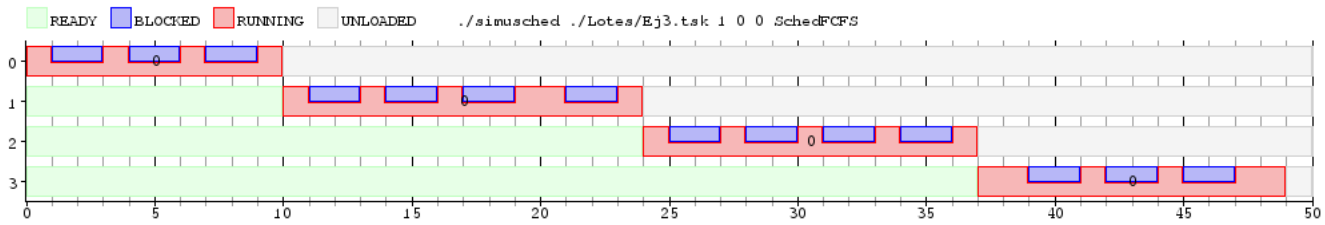


*Gráfico*3.1 Scheduler FCFS - 1 core

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 implementación recibe como primer parámetro la cantidad de núcleos y a continuación los valores de sus respectivos quantums. Debe utilizar una única cola global, permitiendo ası la migración de procesos entre núcleos.

- **Ejercicio 5** Disene un lote con 3 tareas de tipo TaskCPU de 70 ciclos y 2 de tipo TaskConsola con 3 llamadas bloqueantes de 3 ciclos de duración cada una. Ejecutar y graficar la simulación utilizando el scheduler Round-Robin con quantum 2, 10 y 50.
  Con un cambio de contexto de 2 ciclos y un solo núcleo calcular la latencia, el waiting time y el tiempo total de ejecución de las cinco tareas para cada quantum. ¿En cual es mejor cada uno? ¿Por qué 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 gráficos de este ejercicio y el anterior, explique las diferencias entre un scheduler Round-Robin y un FCFS.

- **Ejercicio 7** El scheduler SchedMistery fue creado por docentes investigadores de nuestra materia y ha sido destacado en la ultima publicación de ACM - SIGOPS, Operating Systems Review. Desde entonces, numerosos investigadores de todo el mundo nos han contactado para pedirnos su código fuente. Sin embargo, su código no aparece en ninguno de los repositorios de la materia y nadie parece recordar quienes habıan estado detras de su implementación. Se les pide experimentar con dicho scheduler (aprovechando que hemos conseguido el códi go objeto) y replicar su funcionamiento en SchedNoMistery. Graficar como maximo tres lotes de tareas utilizados en los experiementos y explicar en cada uno por separado que características de SchedMistery identificaron con ese lote. Nota: El scheduler funciona para un solo core y toma uno o mas argumentos numéricos.

- **Ejercicio 8** Implemente un scheduler Round-Robin que no permita la migración de procesos entre núcleos (SchedRR2). La asignación de CPU se debe realizar en el momento en que se produce la carga de un proceso (load). El núcleo correspondiente a un nuevo proceso ser´ aquel con menor cantidad de procesos activos totales (RUNNING + BLOCKED + READY).Explique un escenario real donde la migración de núcleos sea beneficiosa y uno donde no (mencione específicamente que métricas de comparación vistas en la materia mejorarıan en cada caso). Disene un lote de tareas en nuestro simulador que represente a cada uno de esos escenarios y grafique su resultado para cada implementación. Calcule y compare en cada gráfico las métricas que menciono.

4.2 Resultados y Conclusiones

4.2.1 Ejercicio 4

Para desarrollar la implementación del scheduler $Round - Robin$ y que éste 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 están 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 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*: el mismo guarda en la posición i el quantum que se dispuso a cada núcleo.

4. Un vector *quantumActual* aquí guardaremos la cantidad de ticks que le quedan al proceso desde que fue cargado en el core.

5. Una lista de *bloqueados*: esta tendrá los procesos que se bloquearon cuando estaban corriendo.

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, misma es desalojada de la cola global, y agregada en una lista de bloqueados. Además, será reseteado el quantum, se le dará inicio a la próxima tarea que se encuentre ready y cuando el sistema operativo nos envie una señal de unblock, la tarea desalojada regresará al final de la cola global.

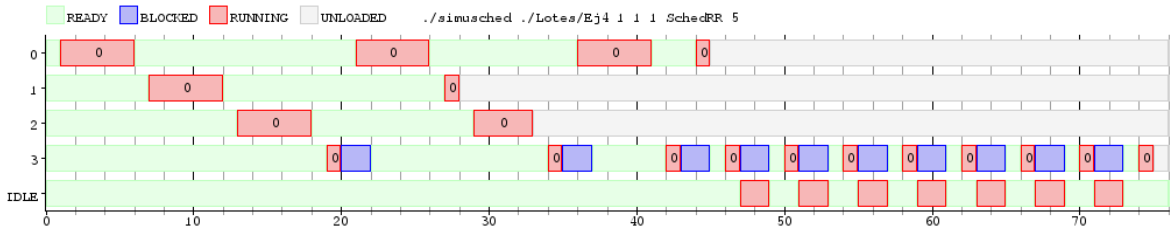


*Gráfico*4.1 – *Lote*1 - Scheduler RR - 1 core - 5 quantum

Se puede ver en el gráfico que cada 5 Ticks de reloj se cambia de tareas de manera cíclica para mantener el fairness lo mas posible con todas las tareas. Salvo cuando una tarea se bloquea que esta es desalojada inmediatamente para no desperdiciar tiempo de computo esperando a que esta de desocupe.

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 70
*2 TaskConsola 3 3 3
```

Obteniendo los siguientes resultados:

Gráfico 5.1 *Lote* 1 - Scheduler RR - 1 core - 2 quantum

Latencia:

- Tarea 0: 1
- Tarea 1: 5
- Tarea 2: 9
- Tarea 3: 13
- Tarea 4: 16
- PROMEDIO: 8.8

Waiting time:

- Tarea 0: 377
- Tarea 1: 380
- Tarea 2: 383
- Tarea 3: 59
- Tarea 4: 62
- PROMEDIO: 252.2

Tiempo total de ejecución:

- Tarea 0: 447
- Tarea 1: 450
- Tarea 2: 453
- Tarea 3: 69
- Tarea 4: 72
- PROMEDIO: 298.2

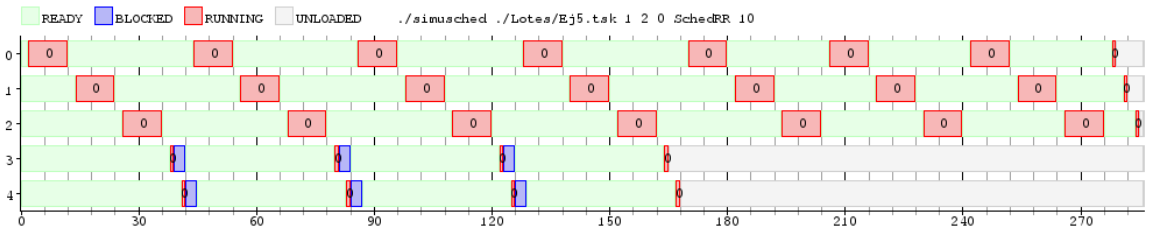


Gráfico 5.2 – *Lote* 1 - Scheduler RR - 1 core - 10 quantum

Latencia:

- Tarea 0: 2
- Tarea 1: 14

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

Waiting time:

- Tarea 0: 209
- Tarea 1: 212
- Tarea 2: 215
- Tarea 3: 155
- Tarea 4: 158
- PROMEDIO: 189.8

Tiempo total de ejecución:

- Tarea 0: 279
- Tarea 1: 282
- Tarea 2: 285
- Tarea 3: 165
- Tarea 4: 168
- PROMEDIO: 235.8

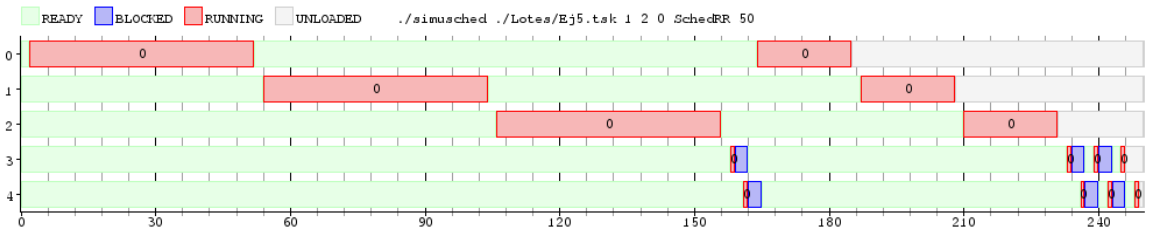


*Gráfico*5.3 – *Lote*1 - Scheduler RR - 1 core - 50 quantum

Latencia:

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

Waiting time:

- Tarea 0: 115
- Tarea 1: 138
- Tarea 2: 161
- Tarea 3: 236

- Tarea 4: 239
- PROMEDIO: 177.8

Tiempo total de ejecución:

- Tarea 0: 185
- Tarea 1: 208
- Tarea 2: 231
- Tarea 3: 246
- Tarea 4: 249
- PROMEDIO: 223.8

- El caso con mejor latencia es el primer experimento, esto se debe al escaso quantum que se le otorga a cada tarea permitiendoles 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 70
*2 TaskConsola 3 3 3
```

Obteniendo los siguientes resultados:

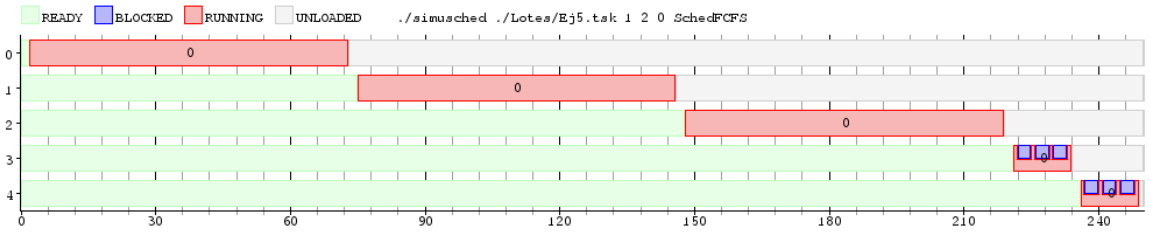


Gráfico 6.1 – *Lote* 1 - Scheduler FCFS - 1 core

A modo de análisis, 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 tenes 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 SchedMistery 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:

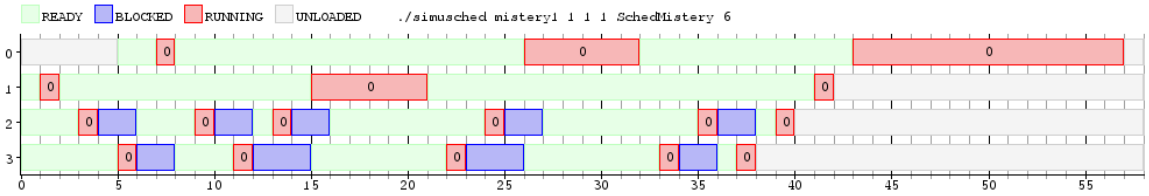


*Gráfico*7.1 – *Lote*1 - Sched Mistery - 1 core - Quantum = 6

De aquí, pudimos deducir que una tarea al bloquearse es desalojada y cuando la misma se desbloquea, el scheduler, le da prioridad para que vuelva a correr.

Con el segundo lote:

```
TaskCPU 20
TaskConsola 20 4
TaskAlterno 1 1 1 1 1 1 15 1 1 1 1
```

Obteniendo lo siguiente:

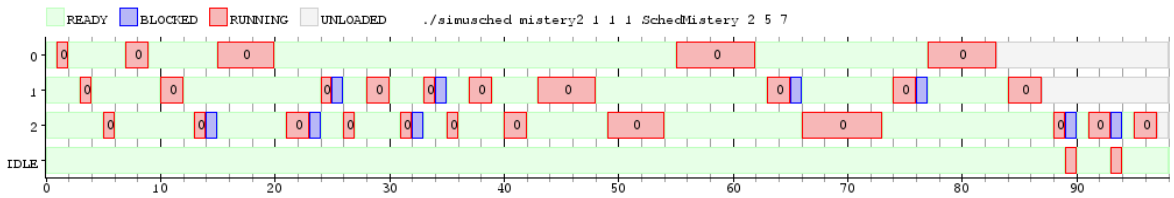


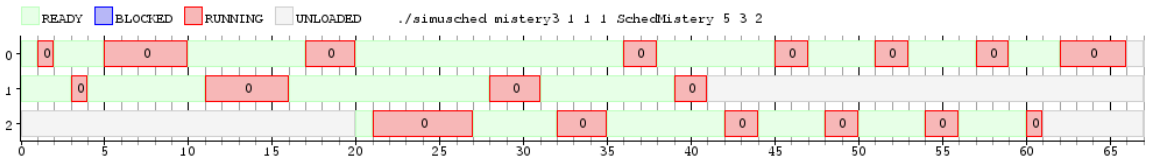
*Gráfico*7.2 – *Lote*1 - Sched Mistery - 1 core - Quantum = 2 - 5 - 7

Con este lote lo primero que se observa es que los parámetros pasados al scheduler van a ser una lista de quantums donde cada tarea va a correr una única vez hasta el útimo quantum de la lista, y a partir de aquí siempre van a correr este. Además esta lista siempre tiene un primer quantum de valor 1.
También observamos que una tarea al desbloquearse vuelve a correr con el quantum anterior de la lista de los mismos. Las tareas cuando se desbloquean se les dá "prioridad" porque la prioridad es dada por el quantum que esta antes en la lista de quantums.

Con el siguiente lote:

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

Obteniendo lo siguiente:



*Gráafico*7.2 – *Lote*2 - Sched Mistery - 1 core - Quantum = 5 3 2

De aquí, pudimos observar, que si una tarea entra más tarde corre los quantums que ya pasaron de la lista. Esto da una noción de prioridad haciendo que estas tareas corran varias veces seguidas antes de que vuelva a correr otra.

En conclusión:

- 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 corren un quantum previo que tiene mayor prioridad)
- Prioridad para tareas más recientes (Corren los quantums que ya pasaron de la lista hasta el quantum actual, esto provoca que corra varias veces antes que sigan corriendo las tareas que entraron antes)

Para desarrollar la implementación del scheduler *No Mistery* y que este funcione de una forma correcta utilizamos una serie de estructuras puntuales.
Las mismas son las siguientes:

1. Un entero *quantumActual* en el cual almacenaremos el valor de quantum que tiene en el momento la tarea que esta corriendo.
2. Un vector de enteros llamado *quantum*, esta secuencia contiene en orden el valor de cada quantum pasado por parámetro.
3. Vector de colas de enteros de nombre *colasXQuantum*. Cada índice de la secuencia es una cola perteneciente a los quantum que se pasaron por parámetro. Estas colas tienen las tareas a ejecutar de cada quantum.
4. Un diccionario *pid_quantum* que dado el pid de una tarea me dice que quantum de la lista es el que tiene que correr.
5. Una función auxiliar llamada *colasVacias* que devuelve un valor booleano si no hay mas tareas en ninguna cola de los quantum.

A continuación explicaremos por partes el código de nuestro scheduler.

Cuando inicializa lo primero que hacemos mediante un ciclo for es guardar los quantums pasados por parámetro dentro de la secuencia *quantum* y al mismo tiempo se van creando colas vacias en la secuencia $colasXQuantum$.

Cuando una nueva tarea es cargada esta se encola en el primer quantum de la lista y se crea una asociación de dicha tarea con el número del quantum de la lista en el diccionario *pid_quantum*.

En el momento en que una tarea se desbloquea nos fijamos cual fue el último quantum que corrió de la lista y si este no es el primero se la encola en la cola del quantum anterior, caso contrario se la encola en el mismo quantum que corrio la última vez. Por último se actualiza en caso de ser necesario la cola actual en el diccionario *pid_quantum*.

En cada tick de reloj nos fijamos si la tarea termina, se bloquea o corre normalmente. En caso de que termine o se bloquee usamos la función $next(cpu)$ (mas adelante explicaremos que hace esta función) para saber cual es la próxima tarea a correr.
Si es un tick normal de reloj nos fijamos, si la tarea que esta corriendo es la IDLE, primero verificamos si hay alguna tarea encolada con la función $colasVacias()$, en caso de haber al menos una devolvemos la próxima a ejecutar. Si no hay tareas seguimos corriendo la IDLE. Si la tarea que se esta ejecutando no es la IDLE restamos en uno el quantum y preguntamos si este llego a 0, en caso de no ser así se devuelve la misma tarea. Si este llego a 0 nos fijamos que quantum de la lista termino de correr, en caso de que no sea el último, se actualiza el dato en el diccionario *pid_quantum* con el siguiente y se encola en la cola correspondiente a este quantum y devolvemos la próxima tarea a ejecutar.

La función $next(cpu)$ busca de todas las colas de los quantums cual es la primera que no este vacía, se saca el primer elemento de la cola y la función lo devuelve. Además guarda en la variable *quantumActual* el quantum de la cola donde extrajimos la tarea.

Por último la función $colasVacias()$ revisa todas las colas de los quantums y devuelve un valor booleano dependiendo si estan todas vacias (devuelve 1) o si hay alguna que contenga al menos una tarea en cuyo caso devuelve 0.

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 núcleo.
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 núcleos 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 más 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 sería, 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:

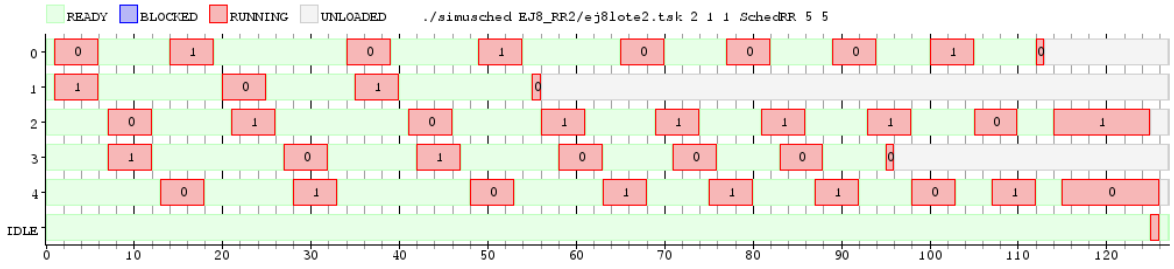


*Gráfico*8.1 – *Lote*3 - Round Robin - 2 core - Quantum = 5 - cambio de contexto = 1

*Gráfico*8.2 – *Lote*3 - Round Robin 2 - 2 core - Quantum = 5 - cambio de contexto = 1

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

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
```

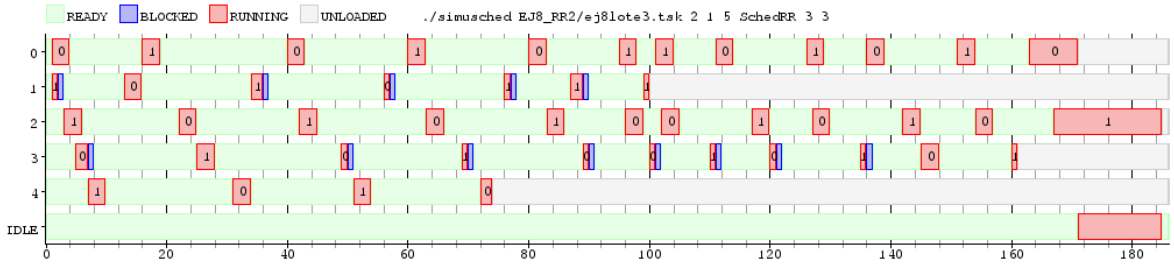


*Gráfico*8.3 – *Lote*3 - Round Robin - 2 core - Quantum = 3 - cambio de contexto = 1

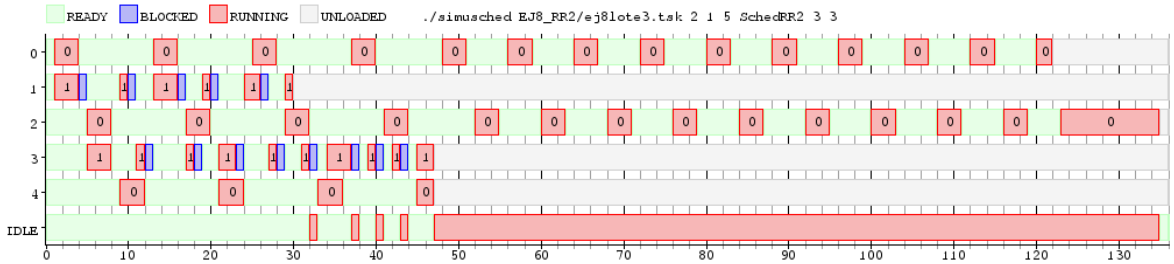


*Gráfico*8.4 – *Lote*3 - Round Robin 2 - 2 core - Quantum = 3 - cambio de contexto = 1

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

Las métricas que más tuvimos en cuenta a la hora de analizar en que casos tiene mejor performance cada scheduler fueron waiting time y turnaround. El turnaround (tiempo de ejecución es lo que mas tomamos en cuenta a la hora de analizar la performance de ambos schedulers, cuanto mas rápido termine de ejecutar una tarea antes va a liberar al sistema para la siguiente tarea. Para analizar que tán rápido se ejecuta una tarea se analiza el waiting time de las tareas en cada caso, una tarea con menos waiting time tiene que esperar menos para volver a correr lo que va a reducir su tiempo de ejecución.
Podemos concluir luego de estas demostraciones que, el Round-Robin original es ámpliamente 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 ámpliamente mejor cuando se utilicen tareas que se bloqueen por un tiempo considerable.

5 Aclaraciones y Bibliografia

5.1 Makefile

Nuestro Makefile fue implementado para poder realizar la experimentación por ejercicio, eliminadolos individualmente y/o todos juntos.
Los nombres son:

- ejercicio1: make ejercicio1 y make clean ejercicio1.
- ejercicio2: make ejercicio2 y make clean ejercicio2.
- ejercicio3: make ejercicio3 y make clean ejercicio3.
- ejercicio4: make ejercicio4 y make clean ejercicio4.
- ejercicio5: make ejercicio5 y make clean ejercicio5.
- ejercicio6: make ejercicio6 y make clean ejercicio6.
- ejercicio7: make ejercicio7 y make clean ejercicio7.
- ejercicio8: make ejercicio8 y make clean ejercicio8.

Al hacer make, se creará una carpeta con el ejercicio y dentro de la misma se encontrara/n el/los gráfico/s Para eliminar todos los gráficos y directorios hacer make cleanTodosLosEjercicios, igualmente este make clean está implementado para eliminar carpetas y/o archivos si estos existiesen o no.

5.2 Bibliografia

- Cátedra de Sistemas Operativos - Clases teóricas y prácticas (1º Cuatrimestre 2016)