



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Scheduling

---

Sistemas Operativos  
Segundo Cuatrimestre de 2015

| Integrante                  | LU     | Correo electrónico   |
|-----------------------------|--------|--|
| Confalonieri, Gisela Belén  | 511/11 | <a href="mailto:gise_5291@yahoo.com.ar">gise_5291@yahoo.com.ar</a>     |
| Mignanelli, Alejandro Rubén | 609/11 | <a href="mailto:minga_titere@hotmail.com">minga_titere@hotmail.com</a> |
| Sabarros, Ian               | 661/11 | <a href="mailto:iansden@live.com">iansden@live.com</a>                 |



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

|   |           |
|---|-----------|
| <b>1. Ejercicio 1 - TaskConsola</b>           | <b>3</b>  |
| 1.1. Algoritmo . . . . .                      | 3         |
| 1.2. Pruebas . . . . .                        | 3         |
| <b>2. Ejercicio 2 - El lote de Rolando</b>    | <b>3</b>  |
| 2.1. Pruebas . . . . .                        | 3         |
| <b>3. Ejercicio 3 - TaskBatch</b>             | <b>4</b>  |
| 3.1. Algoritmo . . . . .                      | 4         |
| 3.2. Pruebas . . . . .                        | 5         |
| <b>4. Ejercicio 4 - Round Robin</b>           | <b>5</b>  |
| 4.1. Representación . . . . .                 | 6         |
| 4.2. Funciones . . . . .                      | 6         |
| <b>5. Ejercicio 5 - Lote para Round Robin</b> | <b>6</b>  |
| <b>6. Ejercicio 6 - Round Robin vs FCFS</b>   | <b>8</b>  |
| <b>7. Ejercicio 7 - Scheduler No Mistery</b>  | <b>8</b>  |
| 7.1. SchedMistery . . . . .                   | 8         |
| 7.2. Pruebas . . . . .                        | 9         |
| 7.3. Representación . . . . .                 | 10        |
| 7.4. Funciones . . . . .                      | 10        |
| <b>8. Ejercicio 8 - Round Robin 2</b>         | <b>11</b> |
| 8.1. Representación . . . . .                 | 11        |
| 8.2. Funciones . . . . .                      | 11        |
| 8.3. Migración vs No Migración . . . . .      | 12        |

## 1. Ejercicio 1 - TaskConsola

Se programó un tipo de tarea llamado `TaskConsola`, que se ocupa de realizar  $n$  llamadas bloqueantes, cada una con una duración al azar comprendida entre los valores  $bmin$  y  $bmax$ .

### 1.1. Algoritmo

La Figura 1 muestra el pseudocódigo de esta tarea.

`TASKCONSOLA( $n, bmin, bmax$ )`

```
1 for  $i \leftarrow 0..n$ 
2    $ciclos\_bloqueo \leftarrow$  valor "random" en  $[bmin, bmax]$ 
3   bloquear durante  $ciclos\_bloqueo$  ciclos
```

Figura 1: Pseudocódigo TaskConsola

Para el cálculo del valor "random" se utilizó la función `rand()` provista por la librería `stdlib` de C++.

### 1.2. Pruebas

Se creó un lote con 3 tareas de tipo `TaskConsola` para correr en el simulador con un scheduler FCFS. La Figura 11 muestra el resultado de dicha simulación.

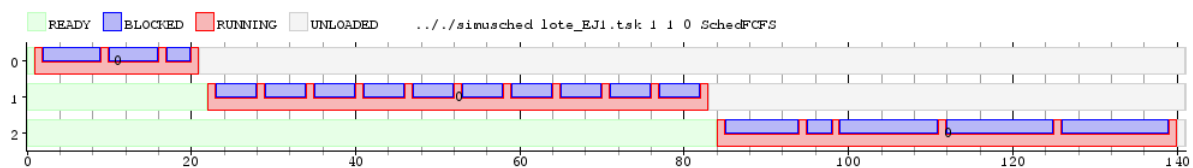


Figura 2: Simulación lote TaskConsola con FCFS, 1 núcleo, 1 ciclos de cs

## 2. Ejercicio 2 - El lote de Rolando

Se escribió un lote de tareas para simular la siguiente situación:

- Correr un algoritmo que hace un uso intensivo de la CPU por 100 ciclos sin realizar llamadas bloqueantes.
- Escuchar una canción, que realiza 20 llamadas bloqueantes con una duración variable entre 2 y 4 ciclos.
- Navegar por internet, que realiza 25 llamadas bloqueantes con una duración variable entre 2 y 4 ciclos.

Estas tareas se ponen a correr en el instante 0, 1 y 2 respectivamente, y Rolando espera que ejecuten en simultáneo.

### 2.1. Pruebas

Se ejecutó el simulador utilizando el algoritmo FCFS para 1 y 2 núcleos, con un cambio de contexto de 4 ciclos. Las Figuras 3 y 4 muestran el resultado de ambas simulaciones.

En el caso de la Figura 3, la latencia de cada proceso es:

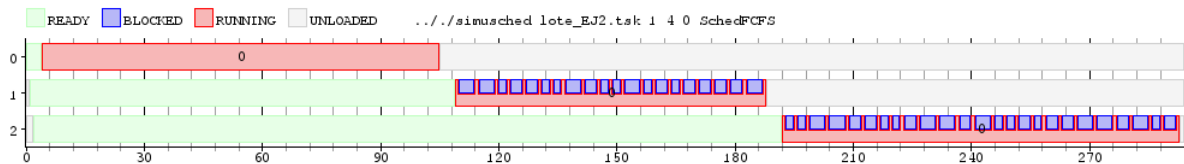


Figura 3: Simulación lote de Rolando con FCFS, 1 núcleo, 4 ciclos de cs

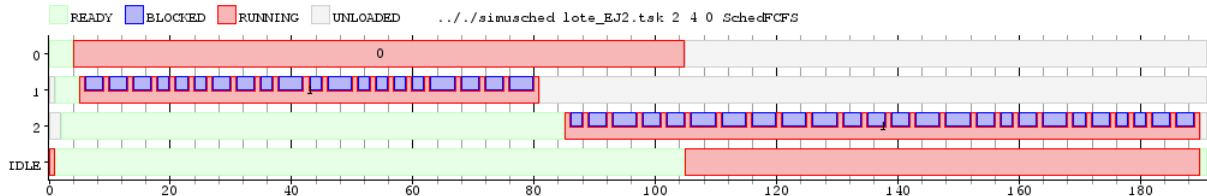


Figura 4: Simulación lote de Rolando con FCFS, 2 núcleos, 4 ciclos de cs

- **Tarea 0:** latencia 4
- **Tarea 1:** latencia 108
- **Tarea 2:** latencia 202

Estos valores nos dan un promedio aproximado de 30,3.

Por otro lado, en el caso de la Figura 4, la latencia de cada proceso es:

- **Tarea 0:** latencia 4
- **Tarea 1:** latencia 4
- **Tarea 2:** latencia 83

Estos valores nos dan un promedio aproximado de 100,6.

En el caso en el que Rolando se viera obligado a utilizar una computadora con un solo núcleo, como se utiliza un Scheduler del tipo FCFServe, no podría escuchar su canción preferida MIENTRAS corre el algoritmo, ya que las tres tareas se ejecutarán secuencialmente. En el caso de poder utilizar una computadora con dos núcleos, podría correrse el algoritmo en uno de ellos y la canción en el otro, y Rolando podría trabajar a gusto.

### 3. Ejercicio 3 - TaskBatch

Se programó un tipo de tarea llamado TaskBatch. Este tipo de tarea realiza *cant.bloqueos* llamadas bloqueantes en momentos elegidos pseudoaleatoriamente, y cada bloqueo dura 1 ciclo. Además, utiliza el CPU durante *total\_cpu* ciclos, incluyendo el tiempo necesario para lanzar las llamadas bloqueantes, pero no el tiempo en el que el proceso permanece bloqueado.

#### 3.1. Algoritmo

La idea de nuestro algoritmo se basa en decidir, a cada ciclo y de manera pseudoaleatoria, si se realiza un bloqueo o no. Para tomar esta decisión vamos a tomar un valor entero “random” entre 0 y 1 (1 para bloquear y 0 para no hacerlo), nuevamente utilizando la función `rand()` de C++.

Como cada llamada bloqueante consume 1 ciclo de utilización de CPU, podemos decir que *total\_cpu* incluye *cant\_bloqueos* ciclos destinados a las llamadas bloqueantes, y el resto son usos “puros” de CPU. Por este motivo, la decisión pseudoaleatoria de bloquear la tarea se realizará  $total\_cpu - cant\_bloqueos$  veces. Una vez realizados todos los usos “puros” de CPU, sólo resta realizar las llamadas bloqueantes necesarias para completar *cant\_bloqueos*.

La Figura 5 muestra el pseudo-código de este algoritmo.

```

TASKBATCH(total_cpu, cant_bloqueos)
1  for  $i \leftarrow 0..(total\_cpu - cant\_bloqueos - 1)$ 
2      bloquear  $\leftarrow$  valor “random” en [0,1]
3      if bloquear == 1  $\wedge$  aún hay bloqueos por hacer
4          bloquear durante 1 ciclo
5          decrementar cant_bloqueos
6      else usar CPU durante 1 ciclo
7  while aún hay bloqueos por hacer
8      bloquear durante 1 ciclo
9      decrementar cant_bloqueos

```

Figura 5: Pseudocódigo TaskBatch

### 3.2. Pruebas

El lote que utilizamos consta de tres tareas, las mismas tratan de mostrar el comportamiento de nuestra TaskBatch. Las características de estas tareas son las siguientes:

1. Posee sólo una llamada bloqueante, para mostrar que no siempre realizará la llamada bloqueante al final, sino que la ubicará en el medio con mayor probabilidad.
2. Posee muchas llamadas bloqueantes, de manera de que las mismas se realicen de forma consecutivas al final con mayor probabilidad.
3. Posee un número de llamadas bloqueantes cerca de un tercio de la cantidad de tiempo de cpu, para mostrar que las llamadas bloqueantes no necesariamente siguen un patrón.

La Figura 6 grafica el lote descrito utilizando un scheduler de tipo FCFS.

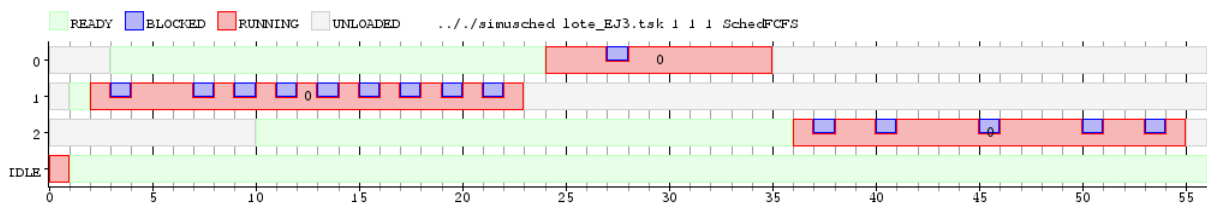


Figura 6: Simulación para TaskBatch, 1 núcleo, 1 ciclo de cs

## 4. Ejercicio 4 - Round Robin

A continuación explicaremos nuestra implementación de un Scheduler Round Robin que permite la migración entre núcleos.

## 4.1. Representación

Hemos representado las tareas con una estructura que contiene:

- un entero para el pid de la tarea
- un entero para el quantum restante de la tarea, en caso de que esté en estado *running*
- un booleano que indica si la tarea está bloqueada o no

Para implementar el scheduler, hemos utilizado los siguientes atributos:

- un arreglo de enteros de tamaño cantidad de cores utilizados, donde cada entero representa el quantum correspondiente a dicho core
- un arreglo de tareas del mismo tamaño, donde cada tarea representa la tarea que actualmente está corriendo en dicho core
- una lista de tareas, correspondiente a las tareas en estado *ready* y *waiting*

## 4.2. Funciones

**Load** Se crea una nueva tarea con el pid indicado y se agrega como último elemento de la lista de tareas *ready* y *waiting*.

**Unblock** Se recorre la lista de tareas *ready* y *waiting* hasta encontrar a la tarea con el pid indicado, y colocarla como “no bloqueada”.

**Tick** Se cuenta con tres casos:

**TICK** Primeramente se decrementa el quantum restante de la tarea actual en el cpu indicado. Si aún tiene quantum para correr, se devuelve su pid. En caso de que se haya terminado su quantum, se evaluará si existe otra tarea en estado *ready*. De ser así, se devolverá el pid de la primer tarea que se encuentre en dicho estado en la lista de tareas. Si no hay otra tarea para correr, sea porque todas las demás se encuentran bloqueadas o porque no existen más tareas, se devuelve el pid de la tarea actual del cpu indicado.

**BLOCK** Se coloca la tarea actual del cpu indicado como “bloqueada”, se la coloca al final de la lista de tareas y se procede a buscar la siguiente tarea a ejecutar. Si no hay más tareas o todas están bloqueadas, se devuelve la constante `IDLE_TASK`. En caso contrario, se devuelve el pid de la primer tarea en estado *ready* que se encuentre al recorrer la lista.

**EXIT** Si no hay más tareas o todas están bloqueadas, se devuelve la constante `IDLE_TASK`. En caso contrario, se devuelve el pid de la primer tarea en estado *ready* que se encuentre al recorrer la lista.

## 5. Ejercicio 5 - Lote para Round Robin

Se diseñó un lote con 3 tareas de tipo `TaskCPU` de 50 ciclos y 2 de tipo `TaskConsole` con 5 llamadas bloqueantes de 3 ciclos de duración cada una. Las Figuras 7, 8 y 9 muestran los resultados de la simulación de este lote con el scheduler Round Robin implementado, con quantum de 2, 10 y 50 ciclos respectivamente.

Para cada caso simulado, se calculó la latencia, el waiting-time y el tiempo total de ejecución para las 5 tareas del lote. El Cuadro 1 muestra los datos obtenidos y el Cuadro 2 muestra los promedios de las 5 tareas para cada quantum.

Basándonos en los promedios calculados, el mejor caso de los simulados para latencia es el que utiliza quantum de 2 ciclos, lo cual puede explicarse dado que teniendo un quantum tan corto, todos los procesos

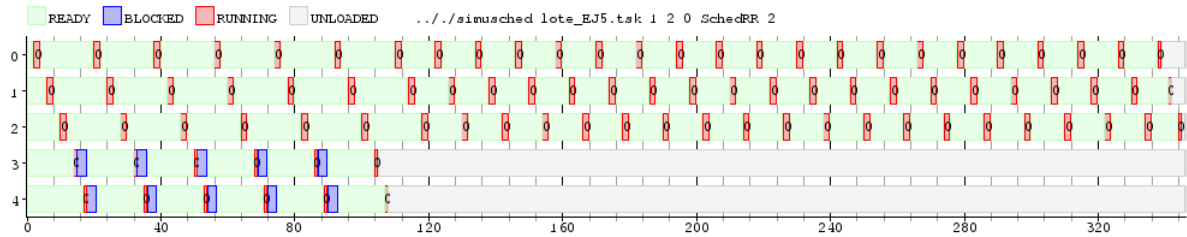


Figura 7: Simulación para SchedRR, 1 núcleo, quantum 2 y 2 ciclos de cs

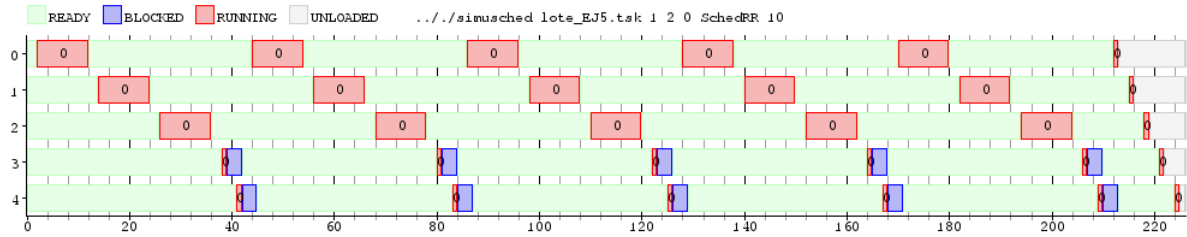


Figura 8: Simulación para SchedRR, 1 núcleo, quantum 10 y 2 ciclos de cs

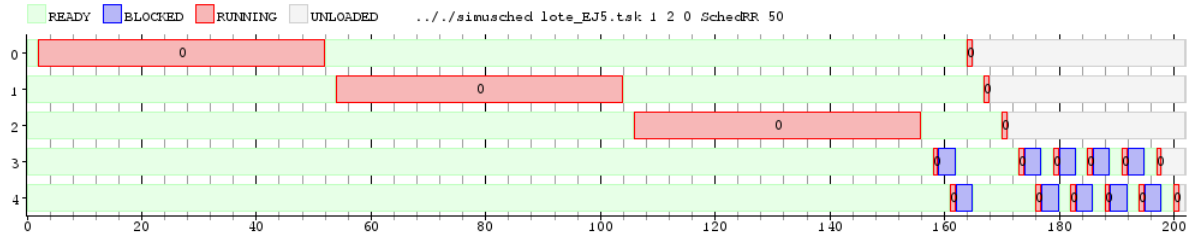


Figura 9: Simulación para SchedRR, 1 núcleo, quantum 50 y 2 ciclos de cs

| Task | Quantum | Latecia | Waiting-time | Total ejecución |
|------|---------|---------|--------------|-----------------|
| 0    | 2       | 2       | 288          | 339             |
|      | 10      | 2       | 162          | 213             |
|      | 50      | 2       | 114          | 165             |
| 1    | 2       | 6       | 291          | 342             |
|      | 10      | 14      | 165          | 216             |
|      | 50      | 54      | 117          | 168             |
| 2    | 2       | 10      | 294          | 345             |
|      | 10      | 26      | 168          | 219             |
|      | 50      | 106     | 120          | 171             |
| 3    | 2       | 14      | 84           | 105             |
|      | 10      | 38      | 201          | 222             |
|      | 50      | 158     | 177          | 198             |
| 4    | 2       | 17      | 87           | 108             |
|      | 10      | 41      | 204          | 225             |
|      | 50      | 161     | 180          | 201             |

Cuadro 1: Datos de simulación - SchedRR

entran en la ronda de ejecución rápidamente. Respecto a waiting-time, el mejor es el caso con quantum de 50 ciclos, y puede deberse a que al tener más ciclos por quantum el tiempo de cambio de contexto tiene menor impacto en el tiempo total de espera. Por último, el mejor tiempo total de ejecución lo tiene también el caso con 50 ciclos de quantum, y estimamos que también tiene que ver el impacto del tiempo invertido en realizar cada cambio de contexto.

Podríamos concluir entonces que, dependiendo de las tareas que se estén ejecutando, si no es primordial que el tiempo de respuesta sea mínimo, utilizar un quantum de 50 ciclos parecería aprovechar de mejor manera los recursos del procesador. Sin embargo, si es prioritario obtener una respuesta sin importar que esto provoque que las tareas demoren mucho tiempo en finalizarse, convendría utilizar el quantum con 2 ciclos.

## 6. Ejercicio 6 - Round Robin vs FCFS

Se ejecutó una simulación con el mismo lote utilizado en la sección 5, pero esta vez con un Scheduler FCFS. La Figura 10 muestra el gráfico de dicha simulación. Los promedios de latencia, waiting-time y tiempo total de ejecución se muestran en el Cuadro 2.

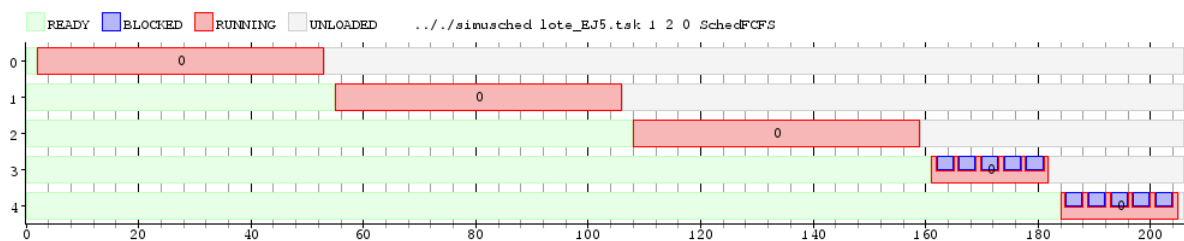


Figura 10: Simulación para FCFS, 1 núcleo y 2 ciclos de cs

| Scheduler | Quantum | Latecia | Waiting-time | Total ejecución |
|-----------|---------|---------|--------------|-----------------|
| RR        | 2       | 9.8     | 208.8        | 247.8           |
| RR        | 10      | 24.2    | 180          | 219             |
| RR        | 50      | 96.2    | 141.6        | 180.6           |
| FCFS      | -       | 102     | 102          | 141             |

Cuadro 2: Promedios - SchedRR y FCFS

Podemos observar en la tabla de promedios del Cuadro 2, que la latencia del FCFS es muy parecida a la del Round Robin con quantum 50, pero los valores de waiting-time y tiempo total de ejecución son significativamente menores. Considerando los resultados obtenidos en el ejercicio 5, podríamos concluir que de tener poca importancia, en cierto contexto de uso, el tiempo que una tarea tarda en empezar a ejecutar, utilizar FCFS tiene una mejor utilización de los recursos del cpu, dejándolo desocupado el menor tiempo posible (con las opciones con las cuales disponemos). Sin embargo, de ser prioritario el tiempo de respuesta de una tarea, sigue siendo más conveniente usar Round Robin con quantum 2.

## 7. Ejercicio 7 - Scheduler No Mistery

### 7.1. SchedMistery

Luego de experimentar arduamente con el Scheduler Mistery, hemos encontrado que implementa un esquema tipo Round Robin con múltiples colas de prioridad, utilizando un solo core.

El nivel de prioriad más alto utiliza un quantum de 1 ciclo, y además el scheduler recibe como parámetros  $n$  valores que representarán el quantum de cada uno de los siguientes  $n$  niveles de prioridad.



Al cargarse una tarea nueva, ésta se coloca en la cola de prioridad más alta. Al terminar su quantum actual, cada tarea baja un nivel de prioridad y se coloca en la cola correspondiente. Por otro lado, cuando una tarea vuelve de un bloqueo, sube un nivel de prioridad y se coloca en la cola correspondiente.

## 7.2. Pruebas

Para experimentar con el Scheduler Mystery hemos creado un nuevo tipo de tarea, a la que llamamos TaskExpMyst. La misma recibe 4 parámetros: *totalcpu*, *lugardebloqueo*, *longbloqueo* y *cant.bloqueos*. Su funcionamiento se basa en hacer un uso de CPU de *totalcpu* ciclos y ejecutar *cant.bloqueos* bloqueos a partir del instante *lugardebloqueo* durando cada una *longbloqueo* ciclos.

Algunos de los lotes con los que experimentaron fueron:

1. Dos usos de CPU de 30 ciclos a partir del instante 0 y un uso de CPU de 30 ciclos a partir del instante 40. Hemos corrido una simulación con este lote para el SchedMyst con 1 ciclo de cambio de contexto y los parámetros 4, 2, 6 y 8. Mediante este experimento hemos notado que los parámetros que recibe el scheduler representan los quantums de cada cola de prioridad (salvo el primer nivel que por defecto es de quantum 1), y que al finalizar el quantum actual, cada tarea utiliza el quantum siguiente, o sea que está corriendo en una prioridad más baja. También se ve que, cuando una tarea ingresa, empieza a correr desde la prioridad más alta y por lo tanto, si la tarea ingresa mientras existen otras tareas activas, éstas no volverán a correr hasta que no haya más tareas con prioridad superior a ellas. En la Figura 11 vemos esta simulación, y podemos notar que las tareas 0 y 1 se van alternando los quantums indicados hasta el momento 44, donde ingresa la tarea 3, ejecuta los primeros 4 quantums (contando el 1) para luego seguir alternándose en la ronda las otras dos tareas.

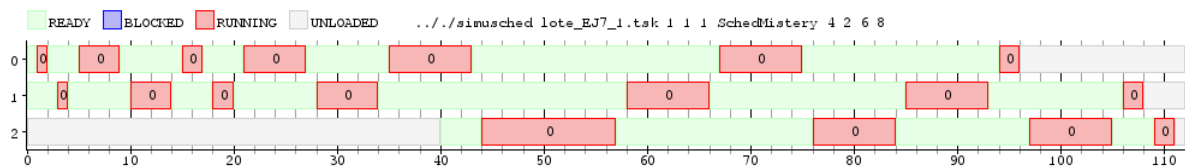


Figura 11: Simulación lote 1 con SchedMystery, 1 ciclos de cs

2. Cuatro usos de CPU de 30 ciclos, dos comenzando en el instante 0, uno en el 35 y otro en el 38. Hemos corrido una simulación con este lote para el SchedMyst con 1 ciclo de cambio de contexto y los parámetros 2, 4 y 6. Mediante este experimento se puede observar que las dos tareas que ingresan últimas se van alternando el CPU hasta llegar al nivel de prioridad de las otras dos tareas que estaban ejecutando desde antes. Esto fue muy importante para lograr determinar que el scheduler funcionaba con colas de prioridad. En la Figura 12 vemos esta simulación, y podemos notar que las tareas 0 y 1 se van turnando los quantums indicados hasta la llegada de las tareas 2 y 3, y luego quedan en estado *ready* hasta que estas últimas llegan a su mismo nivel de prioridad. A su vez se ve claramente cómo las tareas 2 y 3 van pasando por cada nivel de prioridad alternadamente.

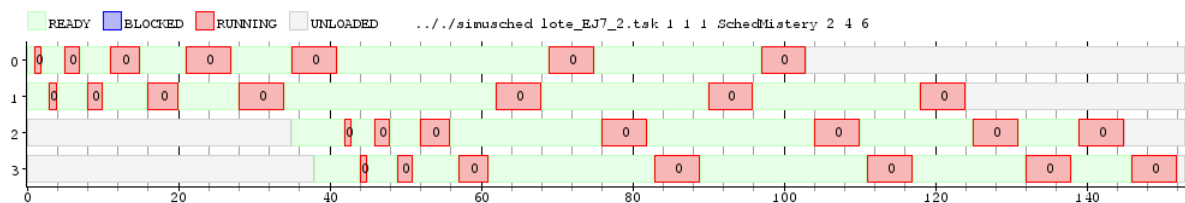


Figura 12: Simulación lote 2 con SchedMystery, 1 ciclos de cs

3. Un uso de cpu de 40 ciclos y, a partir del momento 40, dos tareas de tipo TaskExpMyst con 40 usos de cpu, bloqueándose 2 veces por 2 ciclos a partir del momento 15. Hemos corrido una simulación con este lote para el SchedMyst con 1 ciclo de cambio de contexto. Mediante este experimento se puede observar que cada bloqueo hace que la tarea pase a un nivel de prioridad mayor. En la Figura 13 vemos esta simulación, y podemos notar que las tareas 1 y 2 se bloquean mientras ocupan el nivel de prioridad más bajo, y al salir del bloqueo vuelven a bloquearse, para retomar luego su ejecución en dos niveles superiores de prioridad (se evidencia notando el quantum correspondiente a cada nivel de prioridad).

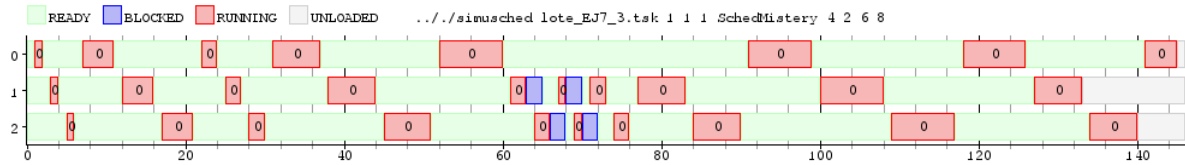


Figura 13: Simulación lote 3 con SchedMystery, 1 ciclos de cs

A continuación explicaremos la implementación realizada para replicar el funcionamiento del Scheduler Mystery.

### 7.3. Representación

Hemos representado las tareas con una estructura que contiene:

- un entero para el pid de la tarea
- un entero para el nivel de prioridad de la tarea
- un entero para el quantum restante de la tarea, en caso de que esté en estado *running*

Para implementar el scheduler, hemos utilizado los siguientes atributos:

- un entero que denota la cantidad de niveles de prioridad
- un arreglo de enteros de tamaño cantidad de niveles de prioridad, donde cada entero representa el quantum correspondiente a dicho nivel
- los datos de la tarea que está corriendo actualmente
- una lista de tareas, correspondiente a las tareas bloqueadas
- un arreglo de tamaño cantidad de niveles de prioridad con una cola de tareas en cada posición, representando las tareas en estado *ready* para cada nivel de prioridad

### 7.4. Funciones

**Load** Se crea una nueva tarea con el pid indicado y prioridad 0, y se agrega al final de la cola de tareas correspondientes a dicha prioridad.

**Unblock** Se recorre la lista de tareas bloqueadas hasta encontrar a la tarea con el pid indicado, se sube un nivel de prioridad (en caso de ser posible) y se agrega al final de la cola de tareas correspondientes a la nueva prioridad.

**Tick** Se cuenta con tres casos:

**TICK** Primeramente se decrementa el quantum restante de la tarea actual. Si aún tiene quantum para correr, se devuelve su pid. En caso de que se haya terminado su quantum, se baja su nivel de prioridad (en caso de ser posible) y se agrega al final de la cola de tareas correspondientes a la nueva prioridad; luego se recorre cada cola de cada nivel de prioridad, de la prioridad más alta a la más baja, y se devuelve el pid de la primer tarea que se encuentre (que eventualmente podría ser la misma que estaba corriendo en este momento).

**BLOCK** Se coloca la tarea actual en la lista de tareas bloqueadas y se recorre cada cola de cada nivel de prioridad, de la prioridad más alta a la más baja, y devolviendo el pid de la primer tarea que se encuentre. Si no hay más tareas en estado *ready* se devuelve la constante `IDLE_TASK`.

**EXIT** Se recorre cada cola de cada nivel de prioridad, de la prioridad más alta a la más baja, y se devuelve el pid de la primer tarea que se encuentre. Si no hay más tareas en estado *ready* se devuelve la constante `IDLE_TASK`.

## 8. Ejercicio 8 - Round Robin 2

A continuación explicaremos nuestra implementación de un Scheduler Round Robin que no permite migración entre núcleos.

### 8.1. Representación

Para las tareas, hemos utilizado la misma estructura que en la implementación del scheduler Round Robin de la sección 4.

Para implementar el scheduler, hemos utilizado los siguientes atributos:

- la cantidad de cores
- un arreglo de enteros de tamaño cantidad de cores utilizados, donde cada entero representa el quantum correspondiente a dicho core
- un arreglo de enteros del mismo tamaño, donde cada entero representa la cantidad de tareas que están asignadas a dicho core
- un arreglo de tareas del mismo tamaño, donde cada tarea representa la tarea que actualmente está corriendo en dicho core
- un arreglo del mismo tamaño conteniendo una lista de tareas en cada posición, donde cada lista contiene a las tareas en estado *ready* y *waiting* para cada core.

### 8.2. Funciones

**Load** Se crea una nueva tarea con el pid indicado, se recorre el arreglo con la cantidad de tareas por cpu, y se agrega como último elemento de la lista de tareas correspondiente al cpu con menos tareas.

**Unblock** Se recorre la lista de tareas *ready* y *waiting* de cada cpu hasta encontrar a la tarea con el pid indicado, y colocarla como “no bloqueada”.

**Tick** Se cuenta con tres casos:

**TICK** Primeramente se decrementa el quantum restante de la tarea actual en el cpu indicado. Si aún tiene quantum para correr, se devuelve su pid. En caso de que se haya terminado su quantum, se evaluará si existe otra tarea en estado *ready*. De ser así, se devolverá el pid de la primer tarea que se encuentre en dicho estado en la lista de tareas correspondiente al cpu indicado. Si no hay otra tarea para correr, sea porque todas las demás se encuentran bloqueadas o porque no existen más tareas, se devuelve el pid de la tarea actual del cpu indicado.

**BLOCK** Se coloca la tarea actual del cpu indicado como “bloqueada”, se la coloca al final de la lista de tareas del cpu indicado y se procede a buscar la siguiente tarea a ejecutar. Si no hay más tareas o todas están bloqueadas, se devuelve la constante `IDLE_TASK`. En caso contrario, se devuelve el pid de la primer tarea en estado *ready* que se encuentre al recorrer la lista correspondiente.

**EXIT** En primer lugar, se decrementa la cantidad de tareas del cpu indicado. Si no hay más tareas o todas están bloqueadas, se devuelve la constante `IDLE_TASK`. En caso contrario, se devuelve el pid de la primer tarea en estado *ready* que se encuentre al recorrer la lista.

### 8.3. Migración vs No Migración

La migración entre núcleos puede ser beneficiosa o no dependiendo del contexto de uso. Por este motivo, mostraremos un escenario factible en el que la migración entre núcleos es conveniente y otro caso en el cual no lo es. Ambos casos serán planteados sin llamadas bloqueantes para una mejor comprensión. Además, mostraremos cada escenario a través del simulador, considerando un cambio de contexto de 1 ciclo, 2 cores y un costo de migración de 4 ciclos.

En un caso en el cual se tienen algunas tareas que hacen un uso de CPU por un tiempo extenso y otras que lo utilizan durante poco tiempo, si no se utiliza migración entre núcleos podría ocurrir que las tareas extensas estén corriendo en un core y las demás en otro. La Figura 14 muestra un ejemplo en el que ocurre esto, y podemos notar que el núcleo 1 rápidamente queda ocioso, mientras que el núcleo 0 sigue corriendo las tareas más pesadas. Por el contrario, la Figura 15 muestra el mismo caso pero utilizando migración, y es evidente cómo se hace un mejor uso de ambos núcleos y, por lo tanto, el tiempo total de ejecución de las tareas más extensas se reduce considerablemente. Si tomamos el tiempo total de ejecución de cada tarea en cada caso, obtenemos para la Figura 14 un promedio de 54 ciclos, mientras que en la Figura 15 tenemos un promedio de 32 ciclos. Además, en el primer caso el núcleo 1 se mantuvo ocioso durante 78 ciclos, mientras que en el segundo sólo lo hizo durante 1 ciclo.

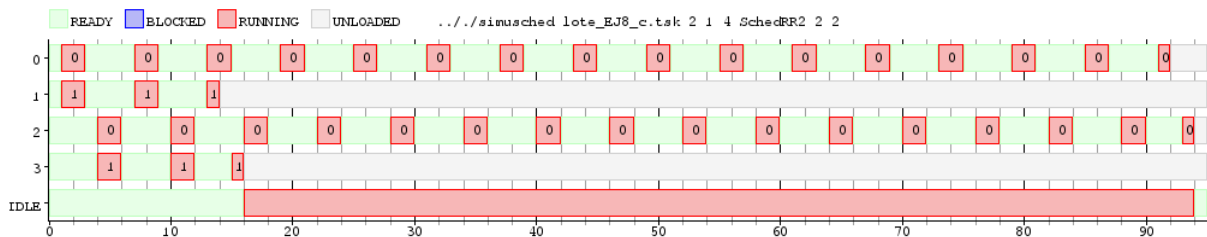


Figura 14: No migración de núcleos no beneficiosa

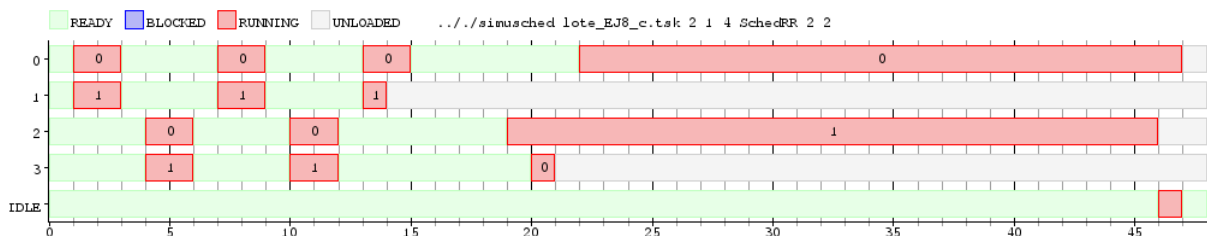


Figura 15: Migración de núcleos beneficiosa

Otro caso que podría darse es que se pongan a ejecutar varias tareas cortas de manera secuencial. Considerando que la migración entre núcleos suele ser bastante costosa, si se requieren numerosas migraciones hasta que cada tarea termina, el waiting-time de ellas se incrementaría de manera considerable, lo cual puede ser poco conveniente dado que se trata de tareas de corta duración. Esto puede verse en la

Figura 16, mientras que, como muestra la Figura 17, no migrar estos procesos permite que terminen en menor tiempo y, por lo tanto, disminuya su waiting-time. Observando la Figura 16 podemos determinar un waiting-time promedio entre todas las tareas de 30,17, mientras que el mismo valor en la Figura 17 es de 14.

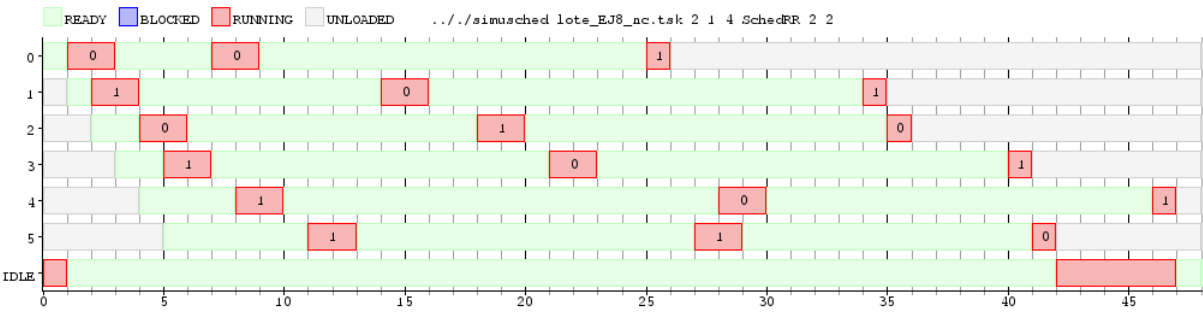


Figura 16: Migración de núcleos no beneficiosa

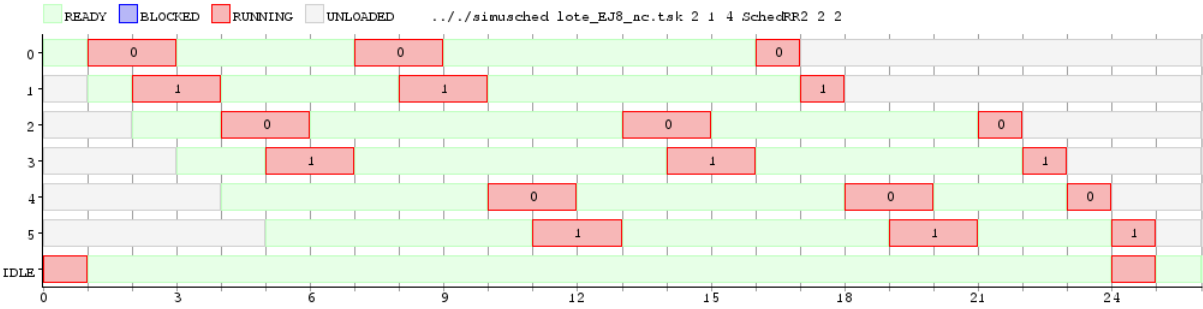


Figura 17: No migración de núcleos beneficiosa