DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Sistemas Operativos

Integrante	LU	Correo electrónico
Salinas, Pablo	456/10	salinas.pablom@gmail.com
Oller, Luca	667/13	ollerrr@live.com
Ituarte, Joaquin	457/13	joaquinituarte@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. Introducción

En este informe explicaremos el desarrollo del código realizado para resolver los problemas del Trabajo Práctico 1 y los experimentos realizados.

El informe consta de 8 ejercicios, tratándose estos de scheduling y el funcionamiento de las distintas políticas de desalojo, el rendimiento simulado obtenido de 1 versus muchos cores y el análisis de gráficos de Gannt.

Los problemas serán resueltos en lenguaje C++, utilizando el simulador simusched provisto por la cátedra.

2. Ejercicios

2.1. Ejercicio 1

En este ejercicio nos piden implementar una función llamada `taskConsola`, la cual simula una tarea interactiva que realiza varias llamadas bloqueantes de una duración al azar. La función tiene como parámetros de entrada el `pid` del proceso que realiza las llamadas bloqueantes y un vector de enteros con 3 elementos. Estos elementos son, en orden, un entero `n` que indica la cantidad de llamadas que realiza la tarea, y dos enteros `bmin` y `bmax` que indican el rango de duración de las llamadas.

La implementación consistió esencialmente en dos pasos: obtener números aleatorios entre `bmin` y `bmax` y luego utilizar una función provista por la cátedra para realizar las llamadas bloqueantes usando la duración obtenida.

Al número aleatorio lo obtuvimos de la siguiente forma:

- Calculamos el rango de duración que puede tener la llamada: `rango = bmax - bmin`.
- Luego, generamos un número aleatorio `num` con la función `rand()` de la librería de C.
- A este número lo usamos para obtener un valor menor o igual a `rango` obteniendo el resto mediante la operación módulo: `num % (rango +1)`.
- Finalmente, a este resultado le sumamos `bmin` y obtenemos un número pseudo-aleatorio entre los valores `bmin` y `bmax`.

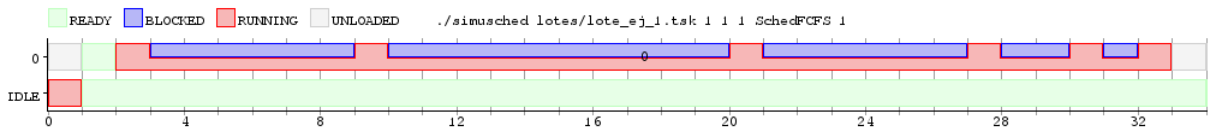
Para realizar las llamadas, realizamos un ciclo de `n` iteraciones, obteniendo en cada una de ellas un número aleatorio (`time`) de la forma explicada anteriormente, y luego utilizamos la función `uso_IO(pid, time)` para realizar las llamadas bloqueantes.

El lote que utilizamos para probar esta tarea fue el siguiente:

```
@1:
TaskConsola 5 1 10
```

Es decir, la tarea `TaskConsola` llega en el instante 1 de la simulación, realiza 5 llamadas bloqueantes y cada una de ellas tiene una duración de entre 1 y 10 ciclos de clock.

El gráfico obtenido es el siguiente:



2.2. Ejercicio 2

En este ejercicio nos piden, dados 4 procesos, analizar la performance de sus ejecuciones con un scheduler FCFS según la cantidad de núcleos que posee la computadora.

Los procesos que se ejecutan son:

- Un proceso que utiliza 500 ciclos de CPU.
- Un proceso que utiliza 10 ciclos de CPU y luego realiza una tarea bloqueante de hasta 4 ciclos (al azar).
- Un proceso que utiliza 20 ciclos de CPU y luego realiza una tarea bloqueante de hasta 4 ciclos (al azar).
- Un proceso que utiliza 30 ciclos de CPU y luego realiza una tarea bloqueante de hasta 4 ciclos (al azar).

Para esto, creamos una tarea llamada TaskDataMining que toma 3 parámetros. En orden, éstos son: el tiempo que utiliza la CPU, la cantidad mínima de tiempo que tarda la llamada bloqueante, y la máxima cantidad de tiempo que tarda la llamada. Luego de las llamadas, los procesos terminan.

Los lotes utilizados son los siguientes:

```
@0:
TaskCPU 500
@1:
TaskDataMining 9 0 4
@1:
TaskDataMining 19 0 4
@1:
TaskDataMining 29 0 4
```

Los gráficos de Gantt obtenidos son los siguientes:

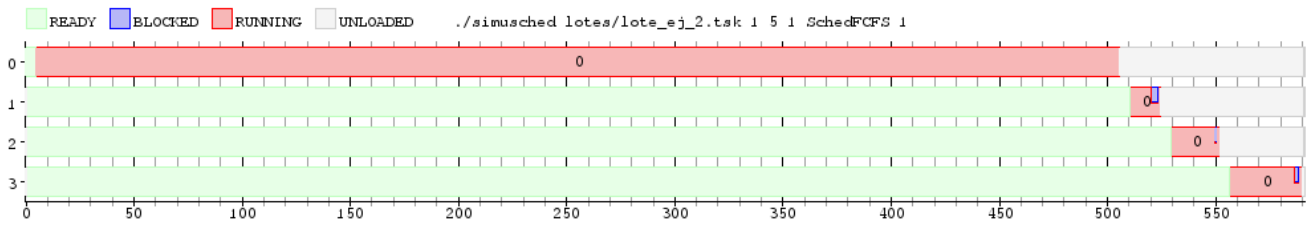


Figura 1: FCFS con 1 núcleo

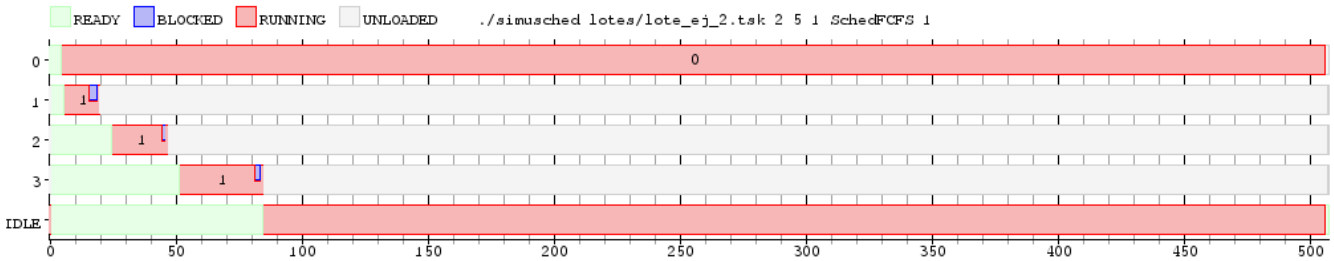


Figura 2: FCFS con 2 núcleos

Figura 3: FCFS con 4 núcleos

Las latencias obtenidas son las siguientes:

Proceso	FCFS 1 núcleo	FCFS 2 núcleos	FCFS 4 núcleos
0	5	5	5
1	510	5	5
2	529	24	5
3	552	50	5

Como puede observarse, la latencia de los procesos disminuye considerablemente con más núcleos, dado que no necesitan esperar a que los otros procesos terminen de ejecutarse (ya que no hay desalojo). Utilizar FCFS con un sólo núcleo podría generar waiting times muy elevados o starvation de los procesos en espera si se ejecuta un proceso que consume mucho CPU o si no termina nunca de ejecutarse. Si bien en estos ejemplos el costo de cambiar de contexto es más alto que esperar a que termine el bloqueo, utilizando este scheduling, si un proceso realiza una llamada bloqueante de larga duración se desaprovecharían los tiempos en los que el proceso permanece bloqueado.

2.3. Ejercicio 3

Para este ejercicio, implementamos el tipo de tarea `TaskBatch`, que recibe como parámetros el tiempo total que la tarea hará uso del CPU (`total_cpu`) y la cantidad de llamadas bloqueantes (`cant_bloqueos`) de 2 ciclos de reloj de duración que deberá ejecutar la tarea.

Como requisitos del enunciado, las llamadas bloqueantes deben ejecutarse en momentos elegidos pseudo-aleatoriamente, y el tick de reloj extra que usan las tareas para realizar la llamada bloqueante debe ser considerado para cumplir el requisito del tiempo total de uso del CPU.

Con esto en mente, nuestra implementación consiste en lo siguiente:

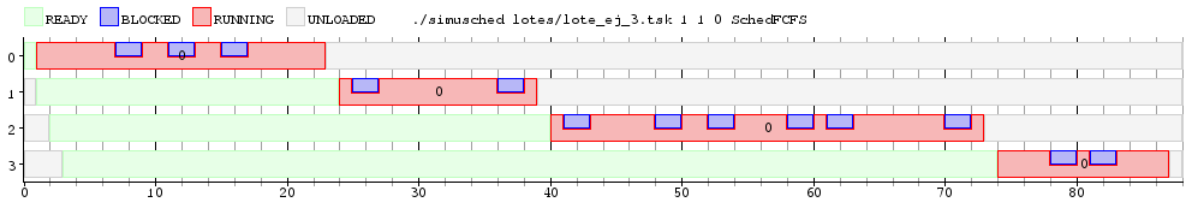
- Generar un array de booleanos de tamaño `total_cpu`, inicializando todas sus posiciones en `false`.
- Generar `cant_bloqueos` números aleatorios distintos entre 0 y `total_cpu-1` (mediante la función `rand()`, teniendo cuidado de evitar repeticiones verificando que la posición generada no estuviera previamente marcada) y setear dichas posiciones del array a `true`.
- Recorrer el array. En las posiciones seteadas a `true`, la tarea ejecutará una llamada bloqueante de duración 2 (`uso_IO(pid, 2)`); en las restantes, la tarea usará el CPU durante 1 ciclo de reloj (`usoCPU(pid, 1)`).

Vale la pena destacar que esta implementación cumple con los requisitos del enunciado: los momentos de bloqueo son elegidos aleatoriamente y en cada uno la tarea se bloquea durante 2 ciclos de clock. Además, la cantidad de tiempo que la tarea hace uso del CPU cumple con ser exactamente igual a `total_cpu` ya que, tanto en el caso en que la posición del array esté marcada como bloqueante o no, se hace uso del CPU durante 1 ciclo de clock (pues, al bloquearse, las tareas consumen 1 ciclo extra de CPU para ejecutar la llamada bloqueante), contabilizando de esta manera `total_cpu` ciclos de clock de uso del CPU.

Por último, mostramos el gráfico pedido por el enunciado: un lote con 4 tareas de tipo `TaskBatch` con distintos parámetros cada una, ejecutadas con el scheduler FCFS.
Lote:

```
@0:
TaskBatch 15 3
@1:
TaskBatch 10 2
@2:
TaskBatch 20 6
@3:
TaskBatch 8 2
```

Diagrama de Gantt resultante:



En el diagrama puede observarse las siguientes cosas:

- Las tareas son atendidas por orden de llegada y no pierden el procesador hasta no haber ejecutado `exit()` (es decir, no hay desalojo).
- Las tareas ejecutan los bloqueos de duración 2 en momentos aleatorios durante su tiempo de existencia (esto se puede verificar volviendo a correr el simulador con el lote presentado y nuestra implementación de `TaskBatch`).
- Las tareas usan el CPU exactamente la cantidad de ciclos indicada por su primer parámetro (descontando el ciclo de clock extra correspondiente a la llamada a `exit()`, que el enunciado no pedía considerar para la cuenta del tiempo total)

2.4. Ejercicio 4

En este punto comentaremos brevemente la implementación realizada de la primera versión del scheduler *Round-Robin*.

La idea del scheduler *Round-Robin* es asignarle una determinada cantidad de ciclos de clock a cada procesador (a esta cantidad la llamaremos el *quantum* del procesador), que representará el mayor tiempo consecutivo que una tarea podrá usar el CPU iésimo antes de ser desalojada de ese procesador. Luego, la idea es recorrer circularmente las tareas activas, dándole a cada una el uso de un procesador que esté libre, durante el quantum que le fue configurado a dicho procesador, repitiendo esta operatoria hasta que todas las tareas hayan terminado con su ejecución.

Con esta idea, desarrollamos nuestro scheduler Round-Robin. Para modelar este scheduler, decidimos usar las siguientes estructuras de datos:

- Una cola `ready_queue` para almacenar los `pid` de las tareas que estén listas para correr.
- Un vector `cpu_quantum` con tantas posiciones como cantidad de CPUs, que se usará para almacenar, en la posición iésima, el quantum del CPU iésimo.
- Un vector `used_quantum`, con tantas posiciones como cantidad de CPUs, que llevará la cuenta del tiempo de CPU usado por el proceso que tiene asignado el procesador iésimo.

En cada tick de reloj, se incrementará el contador del CPU indicado por el parámetro `cpu` de la función `tick()`. En el caso en el que contador llegue al valor del quantum de su CPU, se desalojará a la tarea corriendo en dicho procesador y el contador será reseteado para darle el uso del procesador a la próxima tarea de la cola. El uso de una única cola de tareas nos permitirá migrar a una tarea entre núcleos distintos, ya que el procesador que se le asignará a una tarea a la que le toque correr no será necesariamente el mismo que el procesador en el que la misma tarea ejecutó en su turno anterior (y de hecho, esa información ni siquiera será visible para el algoritmo de scheduling, ya que no se persistirá en ninguna de las estructuras de datos mencionadas).

Para el correcto funcionamiento de la función `tick` se ven reflejados los casos en el cual el proceso que tenga asignado un procesador deba dejar de correr: si el proceso ejecuta una llamada bloqueante, termina con su código (se corresponde con la ejecución de la syscall `exit()`) o el quantum del procesador en el que corría se agotó, el proceso será desalojado y reemplazado por el siguiente de la cola de listos, o por la tarea IDLE si no hubiera ninguno. Además, en el caso en que el quantum del proceso activo se agote, se debe reiniciar el contador del procesador en que estuviera corriendo y, además, volver a encolar a la tarea desalojada en la cola de tareas listas.

2.5. Ejercicio 5

Para este ejercicio, se pide diseñar un lote de tareas de la siguiente forma:

- 3 tareas de tipo `TaskCPU` con 70 ciclos de reloj de uso de CPU.
- 2 tareas de tipo `TaskConsola` que ejecuten, cada una, 3 llamadas bloqueantes, cada una de duración 3.

Ya que no hay requisitos extra sobre el tiempo de llegada de ninguno de los procesos, definimos el lote de la siguiente manera:

```
*3 TaskCPU 70
*2 TaskConsola 3 3 3
```

(Nota: con `b_min = b_max = 3`, las `TaskConsola` se bloquean durante exactamente 3 ciclos)

Para este lote de tareas, se desea experimentar con el scheduler Round Robin, explicado en el punto anterior. Para ello, se dejarán fijos los siguientes parámetros de ejecución:

- Costo de context switch = 2 ciclos de clock.
- Cantidad de núcleos = 1.
- Costo de migración entre núcleos = 0 (notar que, con un solo núcleo, ningún valor de este parámetro afectará en nada al diagrama resultante).

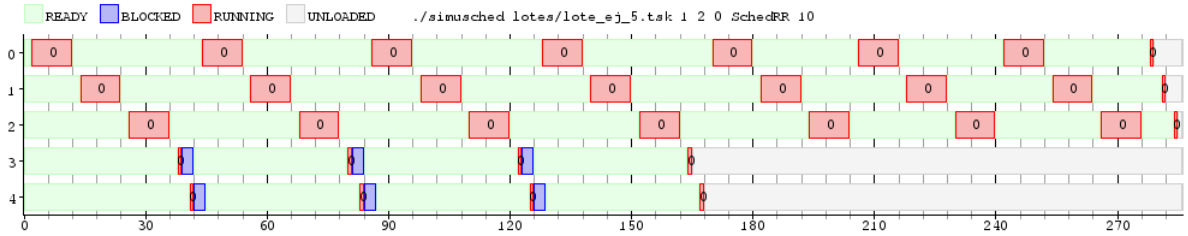
y se tomarán 3 valores distintos para el quantum del único procesador: 2, 10 y 30, buscando comparar las siguientes métricas entre las 3 ejecuciones: *latencia*, *waiting time* y tiempo total de ejecución de todas las tareas. Aprovecharemos este punto para recordar cómo se definen dichas métricas: *waiting time* es la cantidad de ciclos de clock que una tarea pasa en estado *ready*; *latencia* es la cantidad de ciclos de clock que transcurre entre que una tarea pasa por primera vez a estado *ready* y la primera vez que pasa a estado *running*; el tiempo total de ejecución es la cantidad de tiempo transcurrido entre que una tarea pasa a estado *ready* por primera vez y que termina su ejecución mediante la syscall `exit`.

- Con `quantum = 2`, se obtiene el siguiente diagrama:

Y las métricas obtenidas (ordenadas por tarea) son las siguientes:

Id Tarea	Waiting Time	Latencia	Tiempo total
0	376	2	447
1	379	6	450
2	382	10	453
3	65	14	69
4	68	17	72

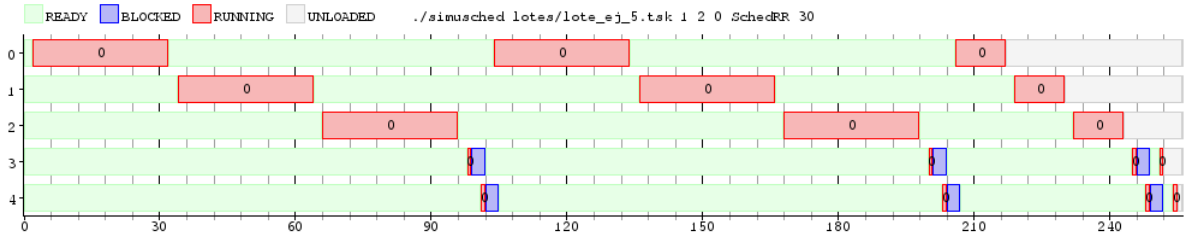
• Con `quantum = 10`, se obtiene el siguiente diagrama:



Y las métricas obtenidas (ordenadas por tarea) son las siguientes:

Id Tarea	Waiting Time	Latencia	Tiempo total
0	208	2	279
1	211	14	282
2	214	26	285
3	152	38	165
4	155	41	168

• Con `quantum = 30`, se obtiene el siguiente diagrama:



Y las métricas obtenidas (ordenadas por tarea) son las siguientes:

Id Tarea	Waiting Time	Latencia	Tiempo total
0	146	2	217
1	159	34	230
2	172	66	243
3	239	98	252
4	242	101	255

A continuación, analizaremos qué métricas son mejores en cada caso y trataremos de explicar por qué.

• *Waiting time:* En los resultados obtenidos observamos que, a medida que se agranda el quantum asignado al procesador, el waiting time es cada vez menor en las tareas de tipo `TaskCPU` (intensivas en el uso del procesador), pero es cada vez mayor en las de tipo `TaskConsola` (intensivas en el uso de entrada/salida). El primer fenómeno puede explicarse de la siguiente manera: con el valor de quantum más pequeño (2) y un costo de migración de 2 ciclos de clock, lo que está haciendo el procesador la mitad del tiempo es ejecutar los cambios de contexto entre tareas, con lo cual, las tareas que necesitan el procesador durante ráfagas largas son más penalizadas por este comportamiento del sistema. A medida que se van asignando quantums cada vez más grandes, este fenómeno se revierte: las tareas con uso intensivo del CPU, al no ejecutar llamadas bloqueantes, pueden disponer del procesador durante todo el período, lo cual les permite completar su uso del CPU (70 ciclos cada una) esperando menos cambios de contexto que para valores más pequeños del quantum. El segundo fenómeno (el de las tareas intensivas en entrada/salida) es inverso: cuanto más grande es el quantum, más tiempo esperan en estado ready ya que, como este tipo de tareas ejecuta una acción bloqueante (de 3 ciclos de clock de duración) apenas obtiene el uso del procesador, la mayor parte del tiempo el procesador está asignado a las tareas intensivas en CPU.

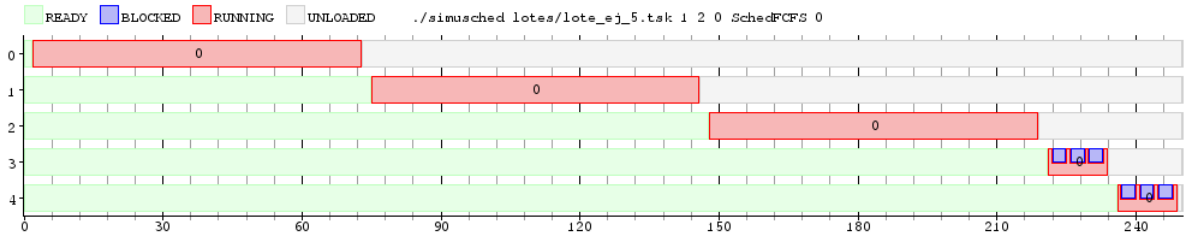
• Latencia: En los resultados obtenidos observamos que, a medida que el valor del quantum crece, el tiempo de respuesta de las tareas es cada vez más grande (esta observación vale para ambos tipos de tareas, a diferencia de la

métrica anterior). Esto es lógico: si cada tarea obtiene el uso del procesador durante períodos más largos de tiempo, las tareas que esperan su turno tendrán tiempos de espera cada vez más largos.

• Tiempo total de ejecución: En este métrica vuelve a darse una situación análoga a la observada en *waiting time*: a medida que el quantum crece, las tareas intensivas en CPU registran tiempos totales cada vez menores, mientras que las intensivas en entrada/salida registran tiempos totales de finalización cada vez mayores. Ya hemos visto que las tareas de tipo `TaskCPU` pasan cada vez menos tiempo en estado *ready* a medida que el valor del quantum crece, lo cual incide directamente en que tengan tiempos de finalización cada vez menores (ya que pasan menos tiempo esperando para poder procesar). Además, un valor grande de quantum hace que cada vez que reciban el procesador, puedan completar una gran porción del procesamiento que deben hacer, con lo cual la cantidad de veces que deben recibir el procesador para poder completar su labor es cada vez menor (y esto además los beneficia en la cantidad de cambios de contexto entre otros pares de procesos que deben esperar hasta recibir el procesador). Por otro lado, este comportamiento en favor de las tareas de tipo `TaskCPU` es totalmente desfavorable para las tareas de tipo `TaskConsola`, ya que las condena a esperar en estado ready todo el tiempo que las tareas intensivas en CPU se pasen procesando (que es cada vez mayor a medida que el valor del quantum crece). Esto explica en gran medida por qué el tiempo total de finalización de las tareas intensivas en entrada/salida va creciendo a medida que el crece el valor del quantum del procesador.

2.6. Ejercicio 6

Para este ejercicio, nos piden graficar el mismo lote de tareas que en el anterior, pero con un scheduler FCFS (first come, first served).
El gráfico obtenido es el siguiente:



A simple vista se ve que en el scheduler FCFS, la CPU ejecuta los procesos en el orden de llegada.
Además, se puede apreciar que la duración de la ejecución del lote de tareas es menor en comparación a las tres ejecuciones con el scheduler Round Robin del ejercicio anterior, lo cual se debe a que la cantidad de cambios de contexto es menor, pues se espera a que cada proceso finalice su ejecución.

La diferencia que hay entre Round Robin y FCFS es que en RR, al tener un quantum, cada proceso tiene su momento de ejecución y si bien el proceso podría no finalizar durante el perìodo que corre, al menos comienza con su ejecución y no se acumulan tantos procesos en la cola esperando el comienzo de su ejecución, como sí podría suceder en el caso de un scheduler FCFS en el que, si el proceso que se está ejecutando tiene un tiempo demasiado largo, al no haber desalojo, los otros procesos se quedan esperando para que comience su ejecución.

Las métricas obtenidas para esta simulación son:

Id Tarea	Waiting Time	Latencia	Tiempo total
0	2	2	73
1	75	75	146
2	148	148	219
3	221	221	234
4	236	236	249

- Waiting time: Podemos observar que el waiting time, es peor para los últimos procesos que llegaron y mejor para los primeros. Esto se debe a que una vez que el proceso comienza su ejecuciòn no se desaloja hasta finalizar la misma, entonces un proceso no debe esperar a su turno, lo cual le permite finalizar antes para los primeros procesos.
- Latencia: La latencia es igual al waiting time para cada tarea ejecutada, esto se debe a que una vez que el proceso comienza su ejecución no es desalojado.

- Tiempo total de ejecución: El tiempo total de ejecución es mejor en general para cada proceso en comparación a los RR, pues al tener menos cambios de contexto, se debe pagar menos veces el costo del context switch, disminuyendo así el tiempo total.

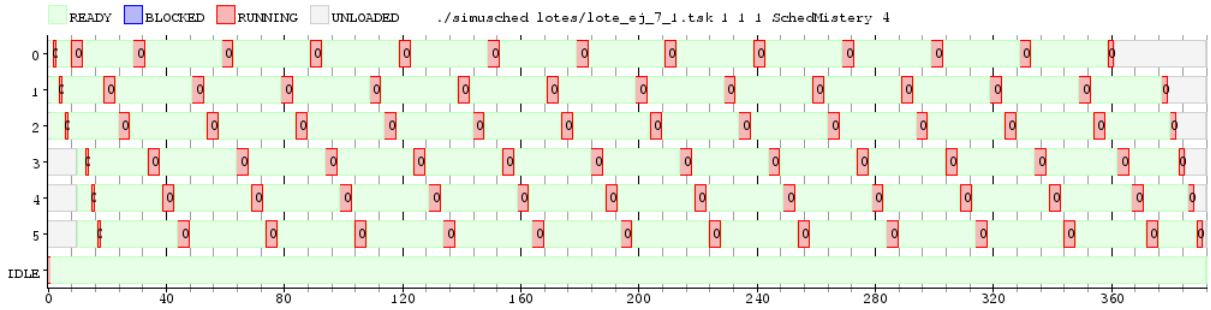
2.7. Ejercicio 7

En este ejercicio nos piden averiguar el comportamiento del scheduler "*SchedMistery*". Como información adicional, sabemos que el scheduler funciona con un solo core y puede tomar varios argumentos numéricos como parámetros de entrada. Como nos piden exhibir a lo sumo 3 lotes utilizados para descubrir lo que hace el scheduler, describiremos los que, a nuestro criterio, fueron los 3 lotes más significativos para ayudarnos a descifrar el algoritmo durante nuestra experimentación.

Lote #1:

```
@1:
*3 TaskCPU 50
@10:
*3 TaskCPU 50
```

Parámetros del scheduler: 4
Diagrama de Gannt obtenido:

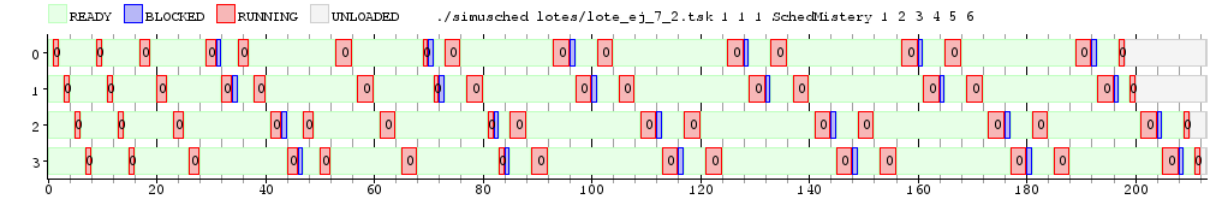


Comportamiento destacado del scheduler: Lo que observamos en este lote es que el primer parámetro de entrada representa a la cantidad de quantum que posee cada tarea y que, cuando una tarea es ejecutada por primera vez, es ejecutada durante un sólo ciclo de clock.

Lote #2:

```
*4 TaskAlterno 5 1 5 1 5 1 5 1 5 1 5 1
```

Parámetros del scheduler: 1 2 3 4 5 6
Diagrama de Gannt obtenido:



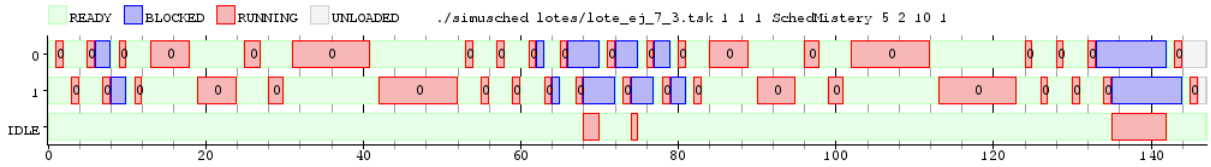
Comportamiento destacado del scheduler: Lo que se puede observar en este lote es que, al agregar más parámetros, el quantum asignado a los procesos y el orden de ejecución cambian. A partir de este lote, pensamos en que cada posición "i" de la entrada corresponde a una cola "i" con un quantum igual al valor de la posición. Viendo el orden y la cantidad de tiempo en el que se ejecutan los procesos, también se puede observar que los procesos se cargan en una cola inicial de quantum 1, y se ejecutan por la cantidad de quantum que corresponda a la cola de la cual provienen. Al terminar de ejecutar, el proceso es encolado en la cola "$i + 1$", y luego se vuelve a ejecutar en un futuro, cuando le corresponda, con el quantum que le corresponde a la cola nueva. Además, se puede ver que hay prioridades en las colas. Es decir, cuando se termina un proceso cualquiera, el siguiente proceso a ejecutar pertenece a la cola "i", con i el valor más chico tal que la cola no este vacía. También observamos que, al realizar llamadas

bloqueantes, los procesos se desalojan y en vez de encolarse en una cola con menor prioridad, se encolan en la cola con 1 nivel de prioridad más alto, de ser posible. Si bien con estos experimentos pudimos deducir el comportamiento general del scheduler, todavía nos falta deducir si la primera vez que se ejecutan los procesos se ejecutan una única vez por un tiempo y luego son encolados a la cola de mayor prioridad, o si bien el scheduler define siempre una cola de 1 quantum de, aparentemente, máxima prioridad.

Lote #3:

```
*2 TaskAlterno 1 2 20 1 0 4 0 3 0 2 20 9
```

Parámetros del scheduler: 5 2 10 1
Diagrama de Gannt obtenido:

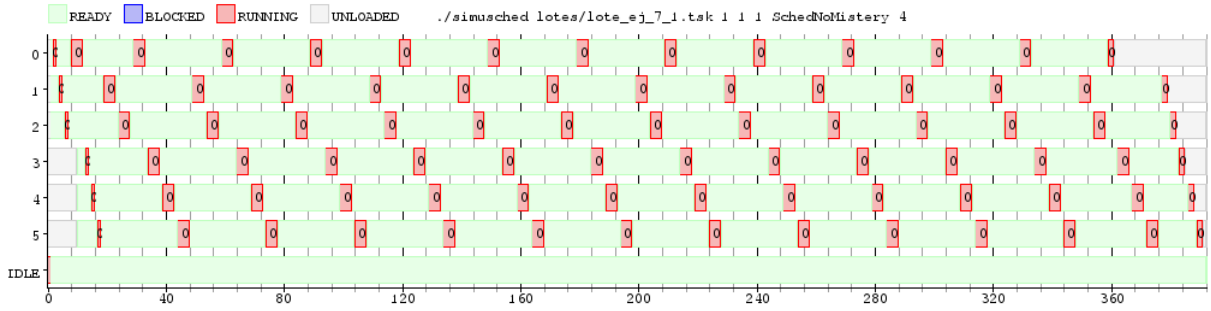


Comportamiento destacado del scheduler: Es este experimento diseñamos un lote para ver el problema con el que nos encontramos con el lote 2. En base al diagrama, se puede deducir que el scheduler genera independientemente de los parámetros a una cola de máxima prioridad a la que se le encolan los procesos al cargarse, y que además dicha cola tiene un quantum de 1 ciclo de clock.

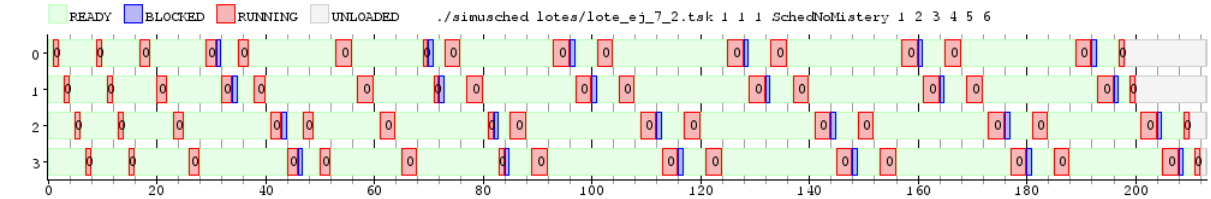
Comportamiento de SchedMistery: Dados los experimentos realizados, podemos deducir que el Mistery Scheduler es un scheduler con varias colas con distintos quantums y prioridades. La cantidad de parámetros de entrada determina la cantidad de colas de tareas. Las colas quedarán configuradas de la siguiente forma: en primer lugar, habrá una cola de máxima prioridad (la llamaremos cola "0") en la cual se le encolan los procesos cuando sean cargados por primera vez, y posee un quantum definido de 1 ciclo de clock. Además de la cola 0, habrá una cola por cada parámetro de entrada. La cola "$i + 1$" tendrá un quantum igual al valor de la posición "i" del vector de entrada. Las prioridades de las colas serán determinadas de la siguiente manera: dadas dos colas cualesquiera, la que tenga el subíndice más bajo será la que tenga mayor prioridad . Los procesos, al ser cargados, comenzarán en la cola de mayor prioridad, donde ejecutarán durante un ciclo de clock, y al terminar de ejecutarse serán encolados en la cola de 1 nivel de prioridad más bajo (si no hay un nivel más bajo se encola en la cola de menor prioridad de todas). Si los procesos realizan una llamada bloqueante, los procesos serán desalojados y se encolarán, al terminar la llamada, en una cola de 1 nivel de prioridad más alto (si el proceso ya provenía de la cola de prioridad más alta, se vuelve a encolar a ésta).

Una vez deducido el algoritmo del scheduler, procedimos a replicar su funcionamiento, obteniendo los mismos resultados para los lotes con los cuales se realizaron los experimentos:

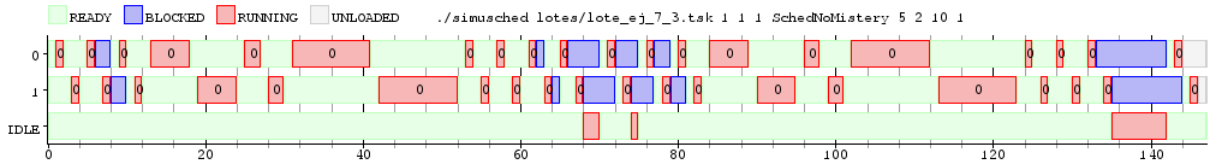
Lote #1: Diagrama de Gannt obtenido:



Lote #2: Diagrama de Gannt obtenido:



Lote #3: Diagrama de Gannt obtenido:



2.8. Ejercicio 8

En este ejercicio, debimos implementar una segunda versión de la política de scheduling Round Robin, cuya diferencia con la primera versión consiste en que no debe permitir la migración de procesos entre cores, para luego experimentar con esta nueva implementación. En primer lugar, explicaremos las estructuras de datos elegidas y los algoritmos para el correcto funcionamiento de este scheduler, y luego pasaremos a la experimentación propiamente dicha.

Para el diseño del scheduler, elegimos las siguientes estructuras de datos:

- Un vector de colas `ready_queue`, con tantas posiciones como núcleos se indique al inicializar el scheduler, para almacenar los `pid` de las tareas que estén listas para correr (separadas por núcleo).
- Un vector `cpu_quantum` con tantas posiciones como cantidad de núcleos, que se usará para almacenar, en la posición iésima, el quantum del núcleos iésimo.
- Un vector `used_quantum`, con tantas posiciones como cantidad de núcleos, que llevará la cuenta del tiempo de CPU usado por el proceso que tiene asignado el procesador iésimo.
- Un vector `active_tasks_count`, con tantas posiciones como cantidad de núcleos, que llevará la cuenta de la cantidad de tareas activas en cada procesador. Este vector se usará para decidir, al momento de cargar una tarea nueva, a qué procesador será asignada.
- Una lista `blocked_tasks` que almacenará pares de la forma `<pid, cpu>`. Su uso será el siguiente: cuando una tarea tarea ejecute una llamada bloqueante, se almacenará en esta lista un par que contendrá su pid como primer componente y el CPU donde estaba corriendo como segunda componente, se la desalojará del procesador y se la removerá de la `ready_queue` correspondiente al CPU donde estaba corriendo al ser desalojada. Luego, cuando esta tarea se desbloquee, se buscará en la lista el único par que tenga el mismo `pid` como primer componente, para volver a encolarla al CPU donde estaba corriendo antes, efectivamente evitando que las tareas migren entre núcleos.

Una vez implementada esta política de scheduling, este ejercicio pide hacer una comparación entre el funcionamiento de las dos versiones de Round Robin, buscando diseñar un lote de tareas por cada uno de los siguientes escenarios:

- Un escenario donde la migración de tareas entre núcleos resulte contraproducente.
- Un escenario donde la migración de tareas entre núcleos resulte beneficiosa.

Para cada uno de los escenarios mostraremos, en primer lugar, la intuición inicial que nos llevó a diseñar el lote de tareas asociado al escenario mostrado, el diagrama de Gantt correspondiente a la ejecución de dicho lote de tareas (para ambas políticas de scheduling), y las métricas que hacen notoria la diferencia de rendimiento entre las dos políticas.

Para el escenario en el cual la migración de núcleos resulta contraproducente, nuestro razonamiento fue el siguiente: el funcionamiento de `SchedRR2` hace que, cuando se produce la carga de una nueva tarea, esta sea asignada al procesador que menos tareas activas tenga. Este balanceo de la carga entre procesadores hace que, a cada momento, todos los núcleos tengan aproximadamente la misma cantidad de tareas activas en su cola de tareas asociada, pero el algoritmo para distribuir las tareas por los núcleos a medida que son cargadas no toma en cuenta ninguna propiedad intrínseca de la tarea que está siendo cargada (por ejemplo, su tiempo total de procesamiento o la cantidad de bloqueos que ejecutará), con lo cual un lote de tareas donde lleguen de manera alternada tareas intensivas en el uso del CPU y tareas que ejecuten acciones bloqueantes largas de entrada/salida podría llevar a una ejecución donde se carguen todas las tareas intensivas en CPU en un procesador y todas las intensivas en entrada/salida en otro, rompiendo"la paridad en el balanceo de la carga entre núcleos. Un escenario real posible para un lote de este estilo puede ser, por ejemplo, un servidor que permita, por un lado, encolar ejecuciones de procesos sobre imágenes grandes y, por otro,

sesiones de usuario interactivas sobre las imágenes ya procesadas.

Con esta intuición en mente, el lote diseñado para este escenario es el siguiente:

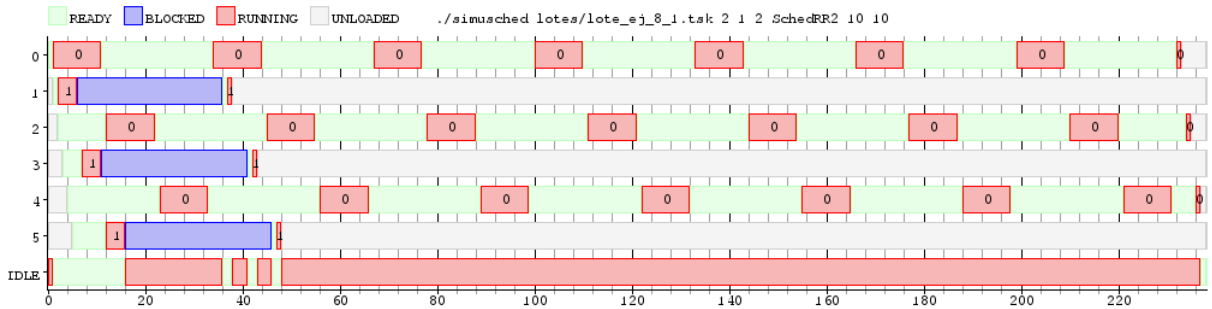
```
@0:
TaskCPU 70
@1:
TaskIO 3 30
@2:
TaskCPU 70
@3:
TaskIO 3 30
@4:
TaskCPU 70
@5:
TaskIO 3 30
```

Al diseñar el lote de esta manera, el comportamiento buscado en `SchedRR2` es el explicado previamente: que todas las tareas intensivas en el uso del CPU sean asignadas al mismo procesador, mientras que todas las tareas intensivas en entrada/salida sean asignadas a otro procesador distinto (en nuestras ejecuciones, configuraremos el simulador con 2 núcleos). Además, si bien el lote diseñado es intencionadamente "malo"(en el sentido de que en el diseño del lote aprovechamos que podemos predecir en qué núcleo se cargará cada tarea) para `SchedRR2`, puede usarse para representar la generalidad de los casos en que la asignación de las tareas a los procesadores termine siendo poco equitativa por las propiedades de las tareas.

De esta manera, las métricas que esperamos que sean mejores en `SchedRR` que en `SchedRR2` son las siguientes:

- Porcentaje de uso del CPU: ya que uno de los procesadores tendrá cargadas las tareas de tipo `TaskIO`, que ejecutarán continuamente bloqueos largos, creemos que este CPU estará desocupado la mayoría del tiempo (esto se corresponderá con la tarea Idle ocupando dicho procesador), bajando drásticamente su porcentaje de uso.
- *Turnaround*: creemos que esta métrica favorecerá al scheduler que permite la migración entre núcleos, especialmente para las tareas de tipo `TaskCPU`, ya que cada tarea de este tipo tendrá que compartir un único núcleo con todas las otras tareas del mismo tipo, con lo cual cada una necesitará más ciclos de clock para terminar.
- *Throughput*: por las consideraciones hechas para la métrica anterior, creemos que esta métrica también debería favorecer al scheduler que permite migración entre núcleos, ya que si cada tarea de tipo CPU necesita más ciclos de clock para terminar, esto implica necesariamente que todo el lote de tareas requerirá más ciclos de clock para terminar de ejecutarse, disminuyendo así la cantidad de tareas que terminan por ciclo de reloj.

Los diagramas de Gantt resultantes para el lote de tareas propuesto son los siguientes:
• Round-Robin sin migración entre núcleos (`SchedRR2`)



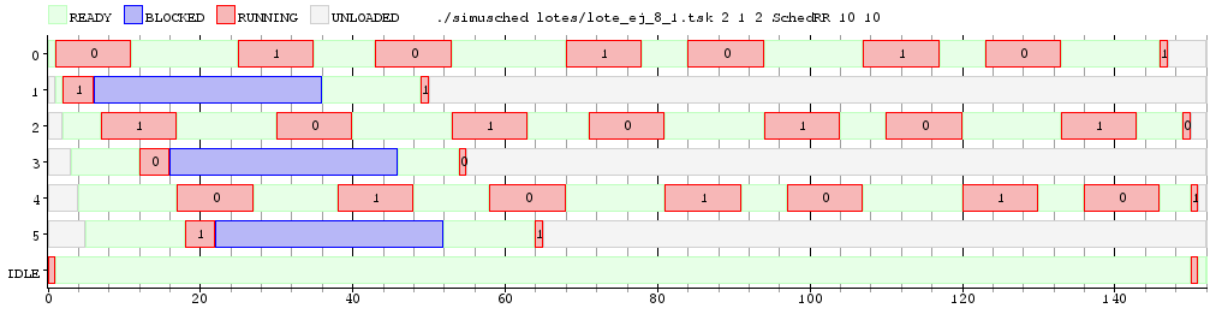
SchedRR2 con 2 núcleos de quantum 10.

Para esta ejecución, obtuvimos las siguientes métricas, agrupadas por tarea:

Id Tarea	Turnaround
0	233
1	37
2	233
3	40
4	233
5	43

Además, las métricas que son globales al sistema son las siguientes:
• Throughput: 6/237. (Nota: indica el promedio entre los 6 procesos que terminan en los 237 ciclos de ejecución del lote)
• Porcentaje de uso del procesador 0: 213/237 = 89.87 %.
• Porcentaje de uso del procesador 1: 15/237 = 6.32 %.

- Round-Robin con migración entre núcleos (`SchedRR`)



SchedRR con 2 núcleos de quantum 10.

Para esta ejecución, obtuvimos las siguientes métricas, agrupadas por tarea:

Id Tarea	Turnaround
0	147
1	49
2	148
3	52
4	147
5	60

Además, obtuvimos los siguientes valores para las métricas globales al lote:
• Throughput: 6/151.
• Porcentaje de uso del procesador 0: 116/151 = 76.82 %.
• Porcentaje de uso del procesador 1: 112/151 = 74.17 %.

Mirando las métricas obtenidas de cada ejecución, podemos decir que nuestras intuiciones iniciales resultaron ser correctas. En primer lugar, los porcentajes de uso del CPU son, en conjunto, drásticamente mejores para el scheduler que admite migración entre núcleos por los motivos explicados previamente. En segundo lugar, el *turnaround* de las tareas intensivas en el uso del CPU también mejora al usar el scheduler con migración entre núcleos, ya que el promedio para las `TaskCPU` con el `SchedRR2` es de 233 ciclos de clock, mientras que para el `SchedRR` es de 147 ciclos de clock (para las tareas de tipo `TaskIO`, vemos que esta métrica empeora, pero esta penalización no es tan significativa, ya que pasan de promediar 40 ciclos de clock a promediar 53, con lo cual el beneficio sobre las tareas de tipo `TaskCPU` supera ampliamente este costo agregado sobre las tareas de entrada/salida). Por último, y en consecuencia con lo obtenido al medir el *turnaround* de las tareas, vemos que el tiempo total que le lleva al simulador finalizar la ejecución del lote disminuye drásticamente, pasando de 237 a 151 ciclos de clock, con lo cual el *throughput* del sistema también es mejor para el scheduler que permite la migración entre núcleos.

Para mostrar un escenario donde sea más conveniente no permitir la migración de tareas entre núcleos (es decir, donde usar el scheduler `SchedRR2` resulte más conveniente) se puede pensar, como primera aproximación, que el costo

de migración entre núcleos puede resultar diferencial entre un scheduler y el otro. Sin embargo, no basaremos solamente en esta idea el análisis de este caso ya que, trivialmente, podríamos reducir el ejercicio a tomar un lote de tareas cualquiera y aumentar tanto el costo de migración entre núcleos que todos los otros tiempos resulten insignificantes al lado de la suma de todos los costos de migración, logrando de esta manera que todas las métricas elegidas dieran mejor para el `SchedRR2`. Sin embargo, sí creemos que el costo de migración es uno de los factores que puede hacer diferencia en la performance al comparar las dos implementaciones de Round Robin, con lo cual tomaremos un costo de migración levemente más alto para este caso que para el anterior (consideramos que 4 ciclos de clock de costo de migración, frente a 2 ciclos de clock del caso anterior, es un aumento razonable que no impide analizar el resto de las condiciones que hacen a un mejor comportamiento de `SchedRR2` comparado con `SchedRR`).

El razonamiento para armar el lote de tareas será análogo al del caso anterior, pero reverso: supongamos la ejecución de un lote de tareas, de manera tal que las tareas se distribuyan de manera más equitativa entre los núcleos que en el caso anterior (donde todas las tareas intensivas en el uso de CPU le eran asignadas al mismo procesador). Una asignación de tareas de este estilo ocasionará, en primer lugar, que ambos procesadores terminen (aproximadamente) al mismo tiempo de procesar sus tareas asignadas, lo cual mejorará el porcentaje de uso de los procesadores para la versión de Round Robin sin migración (respecto del caso anterior). Además, si la carga en el `SchedRR2` es distribuida de manera más equitativa (en la cantidad de CPU total que necesitarán las tareas que se carguen en cada núcleo), cualquier valor positivo para el costo de migración hará que el tiempo total de ejecución del mismo lote con el `SchedRR` sea mayor, ya que cada tarea pagará el costo temporal extra que impone el costo de migración (salvo el caso en que ninguna tarea necesite migrar de núcleo; dejaremos este caso de lado ya que se corresponde a un comportamiento casi idéntico por parte de ambas versiones de Round Robin: las tareas serán ejecutadas circularmente durante el valor del quantum o hasta terminar, sin migrar de núcleo y, por lo tanto, sin pagar el costo de migración).
Con esto en mente, esperamos ver una mejoría las siguientes métricas en `SchedRR2`, respecto de `SchedRR`:

- *Turnaround*: bajo la suposición de que la carga de las tareas está bien distribuida entre los núcleos, cada tarea se ahorrará el costo de migración entre núcleos en `SchedRR2`, con lo cual cada tarea requerirá en total menos ciclos de clock en terminar su ejecución, respecto del mismo lote en la versión de Round Robin con migración entre núcleos.
- *Throughput*: por el item anterior, si cada tarea requerirá menos ciclos de clock para terminar, esto necesariamente implicará que el lote entero requerirá menos ciclos de clock para terminar, lo cual mejorará el promedio de procesos terminados por unidad de tiempo.
- Porcentaje de uso del CPU: Por el razonamiento expuesto previamente, creemos que el porcentaje de uso del CPU mejorará si las tareas son distribuidas equitativamente, ya que en la versión sin migración de procesos no habrá "tiempos muertos.en que una tarea esté siendo migrada de núcleo.

Con las ideas intuitivas explicadas previamente, el lote de tareas diseñado para este experimento es el siguiente:

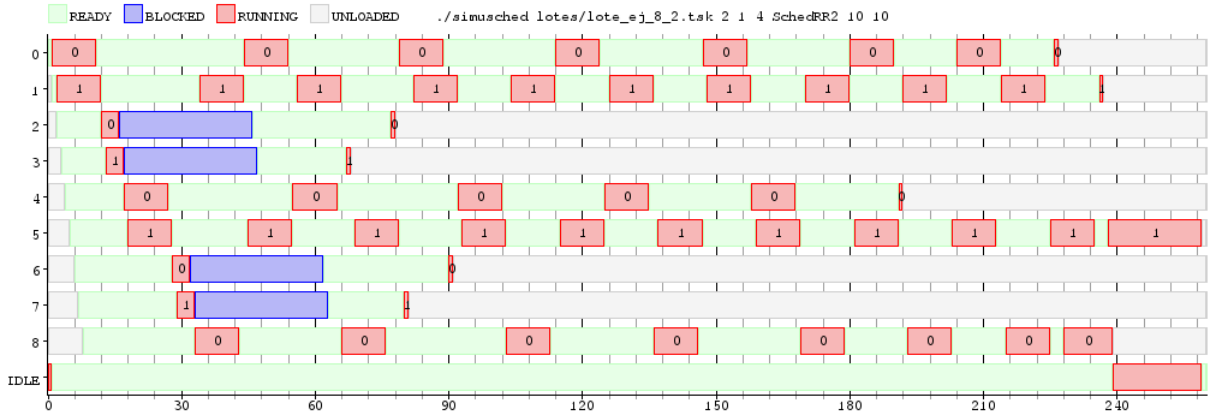
```
@0:
TaskCPU 70
@1:
TaskCPU 100
@2:
TaskIO 3 30
@3:
TaskIO 3 30
@4:
TaskCPU 50
@5:
TaskCPU 120
@6
TaskIO 3 30
@7:
TaskIO 3 30
@8:
TaskCPU 80
```

El lote diseñado cumple con lo expuesto anteriormente: las tareas de tipo `TaskCPU` son cargadas de a dos para que, de cada par, cada una sea asignada a un procesador distinto (al igual que en el caso anterior, configuraremos el

simulador para que corra con dos núcleos), y lo mismo ocurre para las tareas de tipo `TaskIO`. Adicionalmente, para asegurarnos de que en la ejecución con `SchedRR` haya migración de tareas entre núcleos (para incluir el factor del costo de migración entre núcleos al calcular las métricas elegidas), decidimos lanzar un número impar de tareas, condición que, combinada con el resto de los parámetros usados para configurar la corrida, garantiza que haya migración entre núcleos (puede pensarse, intuitivamente, que si las tareas corren de a pares que se ejecutan simultáneamente, una en cada núcleo, un número impar de tareas hará que las tareas no puedan recibir dos veces seguidas el mismo núcleo, con lo cual terminarán migrando entre núcleos).

Los diagramas de Gantt resultantes para el lote de tareas propuesto, junto con las métricas calculadas para cada ejecución son los siguientes:

- Round-Robin sin migración entre núcleos (`SchedRR2`)



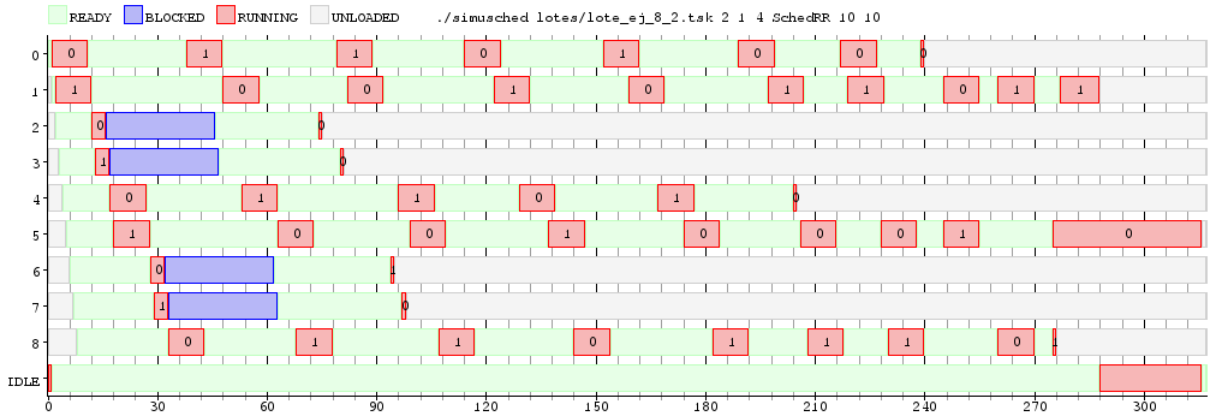
SchedRR2 con 2 núcleos de quantum 10.

Para esta ejecución, obtuvimos las siguientes métricas, agrupadas por tarea:

Id Tarea	Turnaround
0	227
1	236
2	76
3	65
4	144
5	254
6	85
7	74
8	231

Además, obtuvimos los siguientes valores para las métricas globales al lote:
- Throughput: 9/259.
- Porcentaje de uso del procesador 0: 213/259 = 82.23 %.
- Porcentaje de uso del procesador 1: 232/259 = 89.57 %.

- Round-Robin con migración entre núcleos (`SchedRR`)



SchedRR con 2 núcleos de quantum 10.

Para esta ejecución, obtuvimos las siguientes métricas, agrupadas por tarea:

Id Tarea	Turnaround
0	240
1	287
2	73
3	78
4	201
5	311
6	89
7	89
8	268

Además, obtuvimos los siguientes valores para las métricas globales al lote:
- Throughput: 9/316.
- Porcentaje de uso del procesador 0: 234/316 = 74.05 %.
- Porcentaje de uso del procesador 1: 211/316 = 66.77 %.

Observando las métricas obtenidas, podemos decir que nuestras ideas iniciales fueron correctas. Los porcentajes de uso de CPU son mejores para el Round Robin que no permite la migración entre núcleos. Además, en todas las tareas del lote propuesto, salvo la tarea 2, el tiempo de *turnaround* es estrictamente menor, y la diferencia en la tarea 2 es de 76 ciclos contra 73, es decir, una diferencia de 3 ciclos de clock, que resulta insignificante comparada al tiempo total de ejecución del lote en cualquiera de las dos ejecuciones (también es poco significativa si se la compara con la suma de las diferencias entre todas las otras tareas, que favorecen ampliamente al `SchedRR2`). Por último, también se puede notar que el tiempo total de ejecución de todo el lote es mucho menor cuando se usa el scheduler sin migración entre núcleos: 259 ciclos de clock de `SchedRR2` contra 316 de `SchedRR`, lo cual mejora también el *throughput*.

Como conclusión al ejercicio, haremos notar lo siguiente: la conveniencia o no del uso de `SchedRR2` frente a `SchedRR` vendrá dada por el algoritmo utilizado para asignar el núcleo en que se ejecutará una tarea al momento de su carga. Pero, como ningún scheduler puede conocer de antemano cuánto tiempo de CPU requerirá una tarea, podría mejorarse el algoritmo de `SchedRR2` con algún mecanismo que lleve la cuenta de cuánto tiempo llevan las tareas ejecutándose, para tratar de balancear la carga en función del tiempo, pero tratando de no realizar demasiadas migraciones de tareas entre núcleos (esto sería, en definitiva, buscar una solución de compromiso entre permitirle cualquier número de migraciones a todas las tareas -`SchedRR`- y no permitirle ninguna migración a ninguna tarea -`SchedRR2`-).