DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Sistemas operativos
Primer Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Agustina Biscayart	380/06	agustinabiscayart@gmail.com
Carmen Premuzic	408/98	asdlkj.1209@gmail.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. Ejercicio 1

1.1. Desarrollo

Sabiendo que la tarea realiza n llamadas bloqueantes y con una duración al azar entre bmin y bmax, lo que hacemos es generar un número aleatorio n veces. Para esto utilizamos la librería estandar *ctime* la cual nos provee de dos funciones: *rand* y *srand*.
Con la función *rand* generamos un número aleatorio entre *0* y *bmax*, al cual aplicaremos el módulo de la diferencia resultante entre el valor mínimo y el valor máximo tal que, al sumarle nuevamente el mínimo, esté contenido dentro del rango definido como parámetro.
Utilizamos *srand* para la generación de números pseudo aleatorios y así poder generar diferentes sucesiones de resultados en las subsecuentes llamadas de *rand*.
Para la llamada bloqueante utilizamos la función *uso_IO* pasándole como parámetro el pid y el número generado aleatoriamente como cantidad de unidades de bloqueo.

1.2. Experimentación

Para graficar este ejercicio utilizaremos el lote de tareas `task_ej1.tsk`. Este lote de tareas es usado para todos los gráficos y de tres TaskConsola con distintos parametros que inician en distintos momentos:

```
TaskConsola 4 4 10
@4
TaskConsola 2 0 12
@50
TaskConsola 5 1 4
```

La representación del mismo usando el scheduler FCFS es la siguiente:

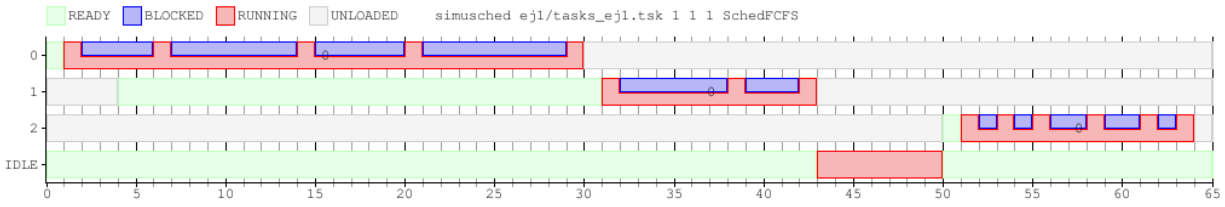

Figura 1: loteEj1.tsk con FCFS

Con este primer gráfico podemos ver como, para cada una de las tres tareas, van apareciendo varias llamadas bloqueantes de una duración diferente las unas de las otras. Considerando que la cantidad para cada una (especificada por el primer parámetro, n) es correcta y que el tiempo de las llamadas está entre los elegidos. Los intervalos no bloqueantes podrían tener un ciclo adicional debido a que cuesta hacer las llamadas bloqueantes y las llamadas de retorno.

Ejecutamos, por segunda vez, en las mismas condiciones para mostrar como varían los tiempos de los bloqueos. El resultado es el siguiente:

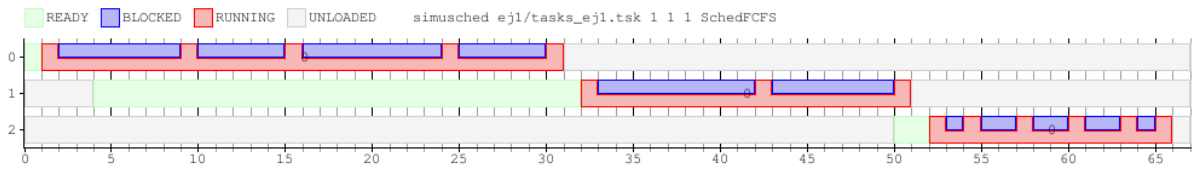

Figura 2: loteEj1.tsk con FCFS segundo intento

2. Ejercicio 2

2.1. Experimentación

El `task_ej2.tsk` contiene lo pedido por enunciado.

```
TaskCPU 500
@10
TaskConsola 50 1 4
@20
TaskConsola 100 1 4
@30
TaskConsola 20 1 4
```

Veamos que tiene un uso de 500 de cpu, y tres TaskConsola que son los tres usuarios que bloquean a tiempo 10, 20 y 30 entre 1 y 4 ciclos.

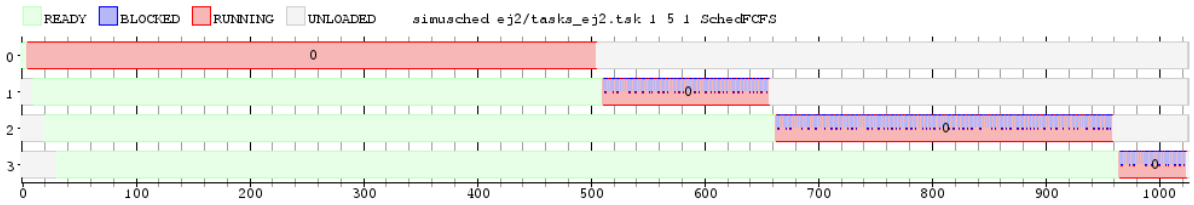

Figura 3: task_ej2.tsk con FCFS y coste cambio de contexto de 5 ciclos con 1 núcleo

Proceso	Latencia
CPU	5
Usuario 1	501
Usuario 2	655
Usuario 3	962

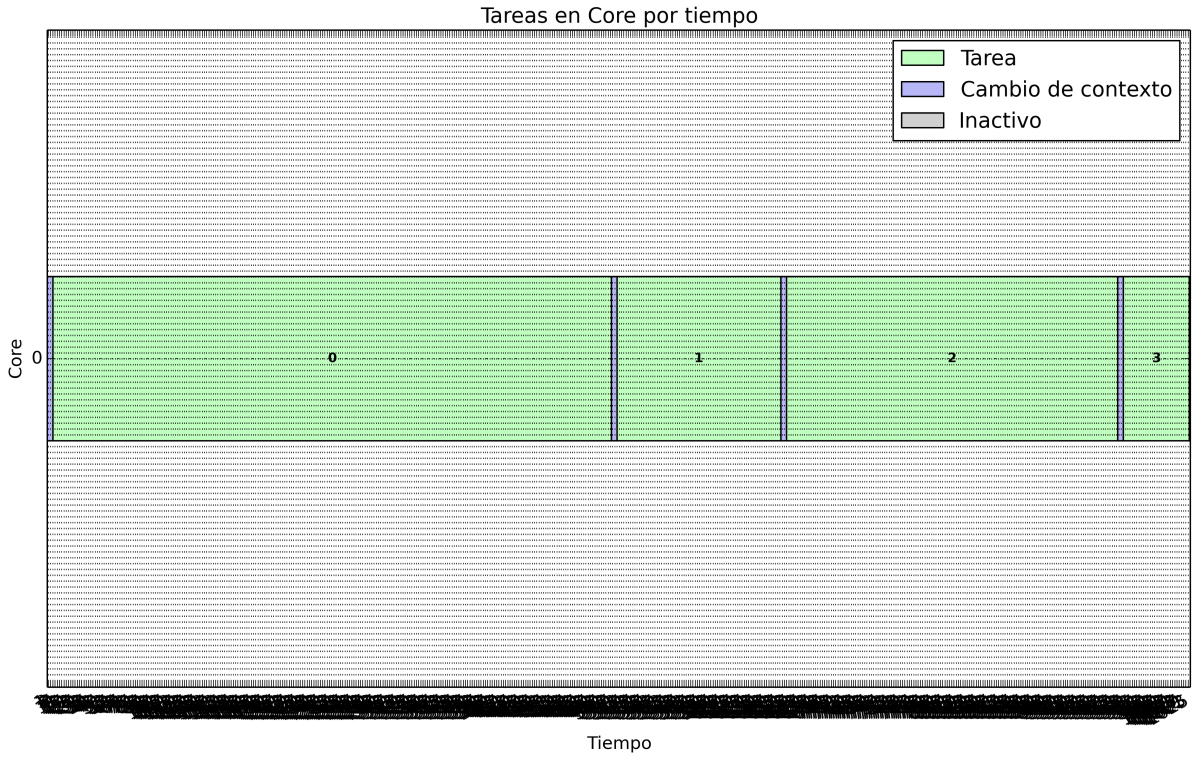

Figura 4: task_ej2.tsk actividad en el núcleo

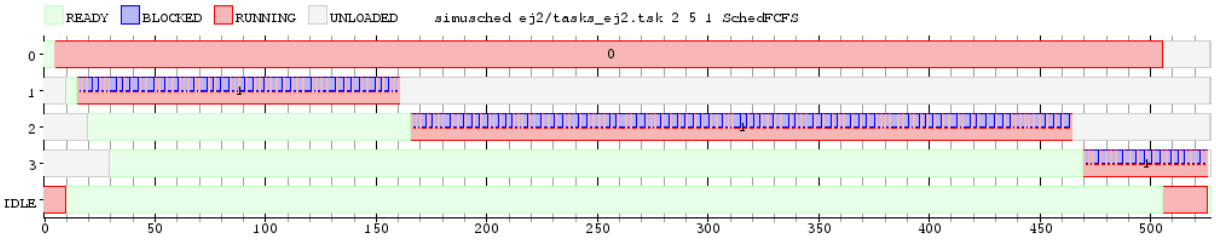

Figura 5: task_ej2.tsk con FCFS y coste cambio de contexto de 5 ciclos con 2 núcleos

Proceso	Latencia
CPU	5
Usuario 1	5
Usuario 2	146
Usuario 3	440

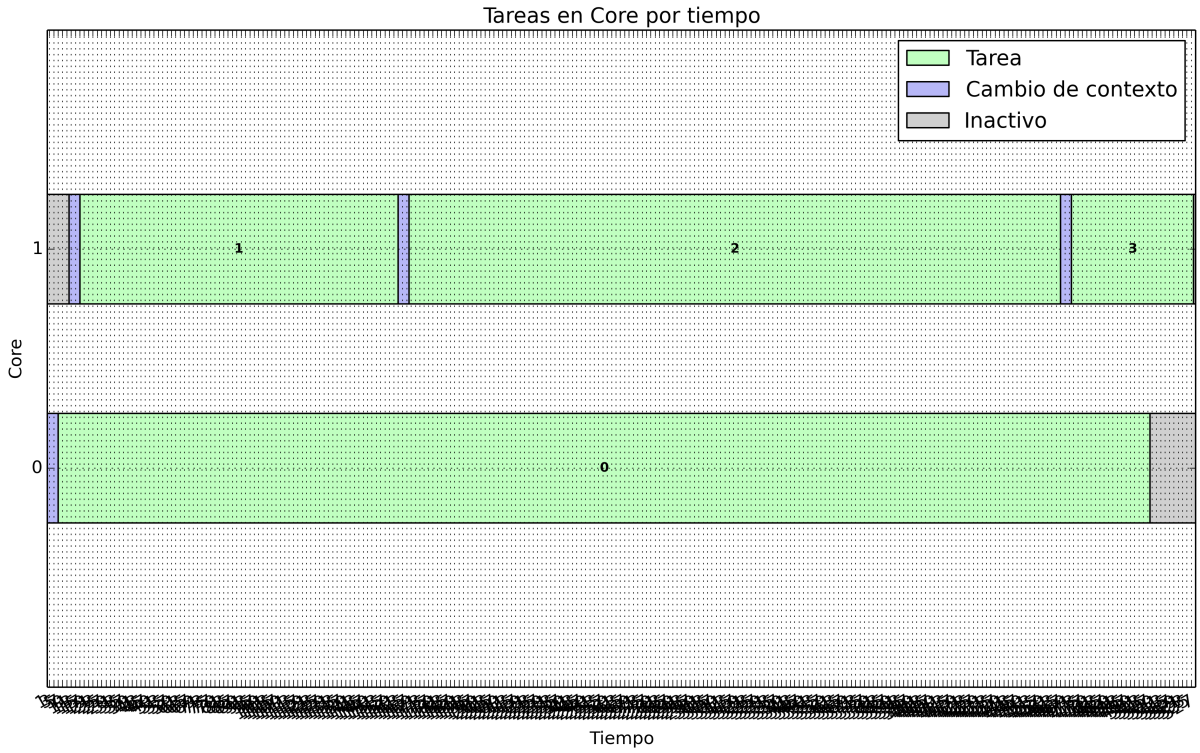

Figura 6: task_ej2.tsk actividad en los 2 núcleos

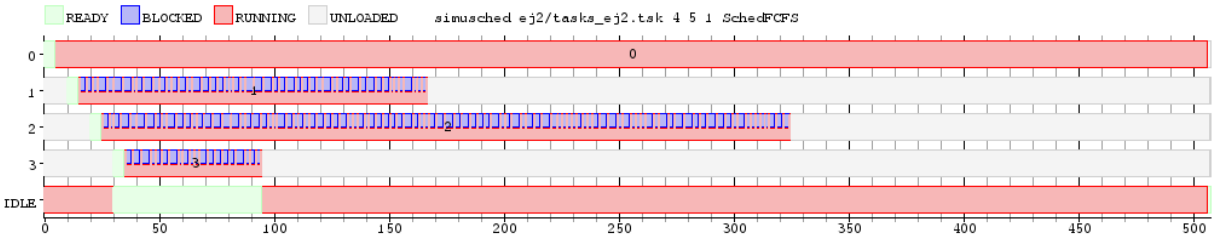

Figura 7: task_ej2.tsk con FCFS y coste cambio de contexto de 5 ciclos con 4 núcleos

Proceso	Latencia
CPU	5
Usuario 1	5
Usuario 2	5
Usuario 3	5

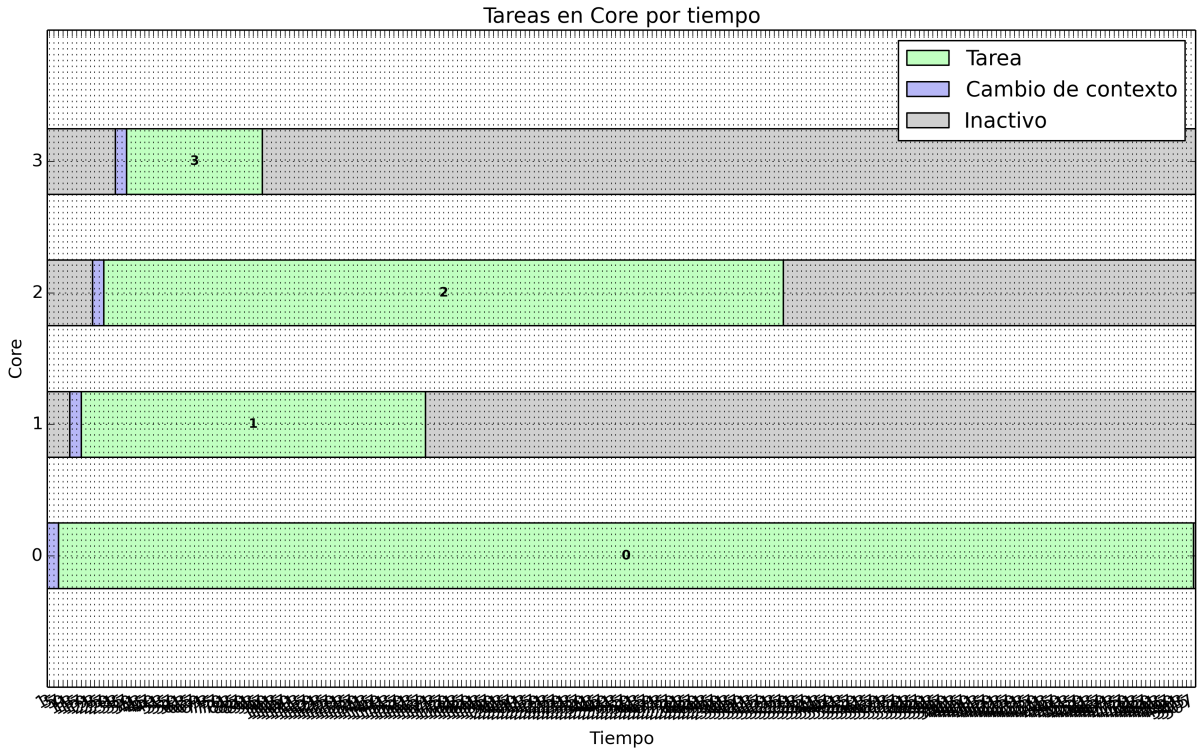

Figura 8: task_ej2.tsk actividad en los 4 núcleos

Hay que aclarar que las latencias podrían variar ya que TaskConsola genera las llamadas bloqueantes con un tamaño aleatorio entre 1 y 4. Podemos ver que en el primer caso (1 núcleo) se debe esperar hasta que la tarea de 500 ciclos termine para que el usuario 1 se pueda conectar remoto y , a su vez, que éste termine para que se pueda conectar el proximo. En el segundo caso (dos núcleos), vemos que se puede ejecutar la tarea de 500 ciclos y conectarse el usuairo 1 a la vez, pero se debe terminar el usuario 1 para poda que se pueda conectar el siguiente, y luego se pueda conectar el usuario 3. La tarea de 500 ciclos ocupa todo un núcleo, y las otras tres tareas ocupan el otro. Por último, para 4 núcleos, cada tarea ocupa uno y la latencia es mínima para todas.

En conclusión, puede llegar a tardar bastante la ejecución de las 4 tareas en un solo núcleo. Una solución podría ser utilizar más cores, u otro mecanimso de scheduling.

3. Ejercicio 3

3.1. Desarrollo

Para poder programar la tarea TaskBatch lo que hacemos es generar un vector con la cantidad de ciclos que va a tener el tiempo de CPU utilizado, guardando el flag `CPU_TASK` para indicar que el sistema debera ejecutar un ciclo de CPU. Luego llenamos las primeras `cant_bloqueos` posiciones con el flag `IO_TASK` para indicar que se debe realizar una llamada bloqueante, debiendo ejecutar exactamente 2 ciclos de reloj.

3.2. Experimentación

Siguiendo lo pedido por enunciado, realizaremos el gráfico del `task_ej3.tsk` usando el FCFS y nos queda lo siguiente:

Lote:

TaskBatch 20 5 //Bloqueo 5 veces en una tarea de 20 ciclos de duración

TaskBatch 7 4 //Bloqueo 4 veces en una tarea de 7 ciclos de duración

TaskBatch 23 11 //Bloqueo 11 veces en una tarea de 23 ciclos de duración

TaskBatch 30 14 //Bloqueo 30 veces en una tarea de 30 ciclos de duración

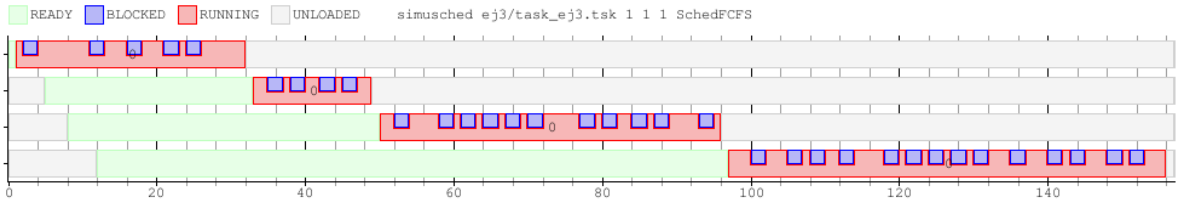

Figura 9: task_ej3.tsk con FCFS con un core

Podemos ver que las tareas ejecutan las cantidades de llamadas bloqueantes indicadas. .

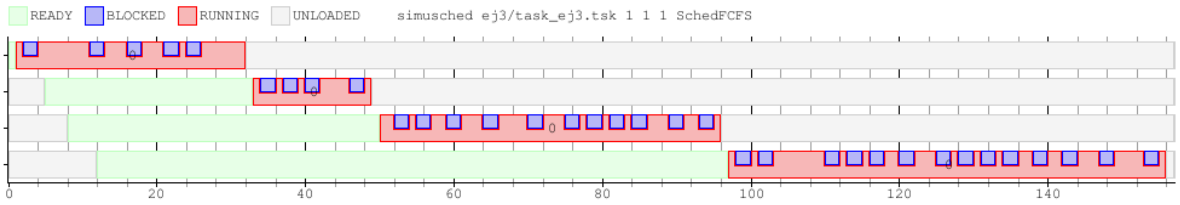

Figura 10: task_ej3.tsk con FCFS con un core (segundo intento)

Para poder asegurar la aleatoriedad del TaskBatch ejecutamos nuevamente con el mismo lote de tareas y observamos que las llamadas bloqueante aperecen en momentos diferentes.

4. Ejercicio 4

4.1. Desarrollo

En este ejercicio implementamos el algoritmo de scheduing Round-Robin (RR). Cada proceso se ejecuta por una cierta cantidad de tics de reloj o quantums, o hasta que termina o es bloqueado. Una vez que un proceso es ejecutado por dicha cantidad de tics de reloj, se pasa a ejecutar el siguiente proceso que no esté bloqueado o terminado. En caso de que un proceso sea bloqueado o termina antes de completar su quantums, se pasa al siguiente proceso.
En la implementación utilizamos tres colas debido a que siempre nos interesa el orden para que el primero en llegar sea el primero en salir.
En la primera, se almacenan todos los procesos que van llegando al scheduler. Esta cola es única y común a todos los procesos sin importar cuántos CPU se estén utilizando.
En la segunda, almacenamos cuánto quantum tiene cada uno de los procesadores, los cuáles son definidos al inicio del programa. Esto solamente se utiliza para reestablecer el valor del quantum que tiene el CPU cuando tiene que procesar otra tarea.
En la tercera, almacenamos el quantum restante que tiene el CPU para la tarea que está realizando actualmente.
Además, realizamos los tres métodos solicitados en el enunciado: *load(pid)*, *unblock(pid)* y *tick(cpu, motivo)*.
En el método *load(pid)* simplemente agregamos el PID a la cola de tareas a procesar.
En el método *unblock(pid)* agregamos la tarea que acaba de desbloquearse en la cola de las tareas listas para ser procesadas.
El método *tick(cpu, motivo)* recibe por parámetro el CPU que requiere el PID y el motivo por el cual el proceso anterior terminó y debe devolver el PID del proceso que se debe ejecutar a continuación. Este motivo puede ser:

1. TICK: la tarea consumió todo el ciclo utilizando el CPU.
2. BLOCK: la tarea ejecutó una llamada bloqueante o permaneció bloqueada durante el último ciclo.
3. EXIT: la tarea terminó (ejecutó return)

Vamos utilizar dos métodos auxiliares: uno llamado next que retorna el próximo pid a ser ejecutado y otro llamado reset que restablece los valores del quantum para un determinado CPU.
Si la tarea actual ha finalizado (EXIT), el algoritmo verifica si en la cola de procesos restantes hay otra tarea a ejecutar, si la hay, devuelve la misma (utilizando el método next mencionado anteriormente), si no, devuelve la tarea IDLE.
Si la la tarea va a ser bloqueada (BLOCK), el algoritmo devuelve la próxima tarea a ejecutar, realizando la misma lógica que en el caso de EXIT.
En el caso de TICK nos encontramos con la problemática de que puede ser llamado tanto por la tarea IDLE como de las tareas que se encuentran dentro de la cola de las tareas a ser ejecutadas.
Si la tarea que invoca el método es una tarea IDLE, el algoritmo verifica si tiene alguna otra tarea a ser ejecutada para retornarla y si no tiene ninguna se sigue ejecutando la IDLE.
Si la tarea que invoca el método es no es la IDLE, el procedimiento es el siguiente: si la tarea tiene quantum restante entonces éste se decrementa y se sigue ejecutando la misma tarea. En caso contrario, se devuelve la próxima tarea siguiendo la misma lógica descripta anteriormente.

5. Ejercicio 5

5.1. Desarrollo

En el ejercicio se pide diseñar un lote con tres tareas de 70 ciclos y dos de 3 llamadas bloqueantes de 3 ciclos de duración. El lote que representamos inicia todas las tareas a la vez

```
TaskCPU 70
TaskCPU 70
TaskCPU 70
TaskConsola 3 3 3
TaskConsola 3 3 3
```

Ejecutamos el lote con el scheduler `SchedRR` para quantums 2, 10 y 30 como se pidió en el ejercicio. Utilizamos un solo cpu con cambio de contexto de 2 ciclos y calculamos la latencia (tiempo desde que se carga la tarea hasta que comienza a ejecutarse), waiting time (tiempo en que la tarea permanece como READY) y el tiempo total de ejecución (tiempo desde que se cargó la tarea hasta que finalizó). Para los cálculos utilizamos los siguientes tiempos (como ciclos) y su forma de obtenerlos del diagrama de Gantt:

ciclos Cantidad de ciclos que permanece una tarea en estado RUNNING.
Para `TaskCPU` de parámetro 70, el valor correspondiente es 71 (70 ciclos de uso de CPU más 1 ciclo para EXIT)
Para `TaskConsola` de parámetros 3 3 3, son 13 ciclos, pues son 3 ciclos de E/S de 3 ciclos de duración más 1 ciclo de CPU cada E/S más un ciclo para EXIT (9 + 3 +1).

inicio Tiempo en el que se carga la tarea. Todas las tareas se cargan en el ciclo 0.

fin Tiempo en el que finaliza la tarea. En el diagrama de Gantt es en el ciclo en el que se realiza el UNLOAD.

latencia Cantidad de ciclos desde que se carga la tarea hasta que comienza a correr por primera vez.

tiempo total Tiempo que tarda la tarea en ejecutarse. En el diagrama de Gantt es la diferencia entre el ciclo en el que termina la tarea y el que se carga: fin - inicio.

waiting time Tiempo en el que la tarea permanece sin hacer nada: tiempo total - tiempo que corre la tarea - tiempo en que la tarea permanece bloqueada.

5.2. Experimentación

En las figuras 11, 12 y 13 se representaron los diagramas de Gantt para los quantums 2, 10 y 30 respectivamente. Se puede ver, cómo a quantums más chicos (figura 11), el scheduler permanece considerable tiempo haciendo cambios de contexto de 2 ciclos. Aún así, en quantums chicos, las tareas de E/S terminan antes que con los quantums más grandes.

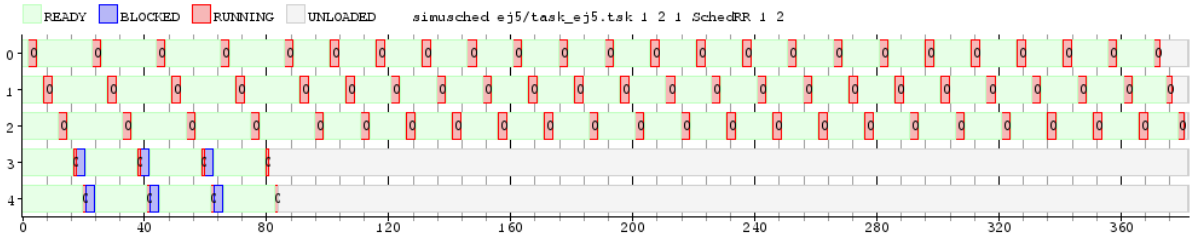

Figura 11: task_ej5.tsk con RR y quantum 2

Para quantums más grandes, todas las tareas terminan muy próximas entre sí, pero a quantums de 30 (figura 13), las tareas de E/S comienzan luego de 95 ciclos (los tres quantums de las tareas de CPU) y podría parecer que el SO no responde.

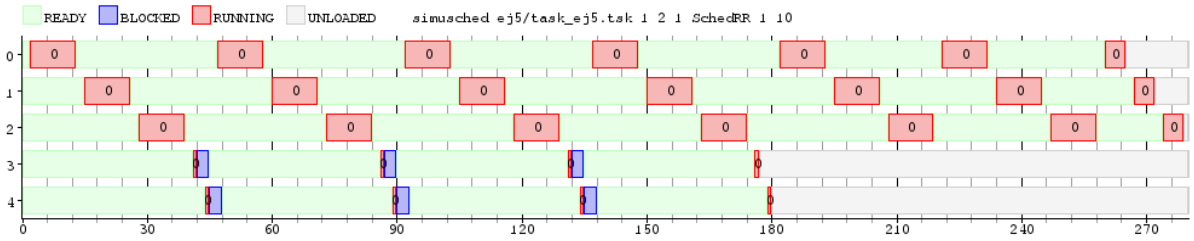

Figura 12: task_ej5.tsk con RR y quantum 10

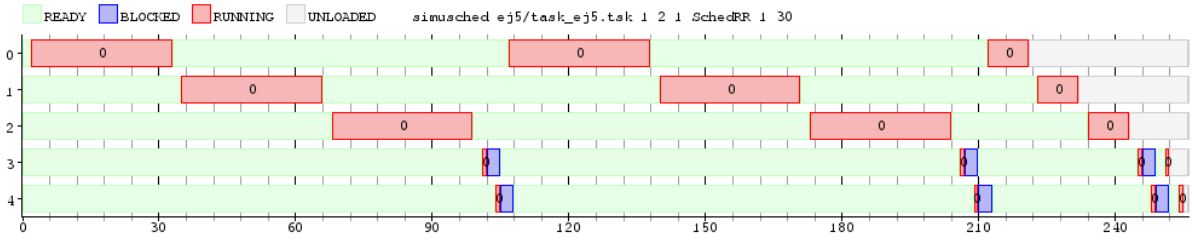

Figura 13: task_ej5.tsk con RR y quantum 50

En las tablas 1, 2 y 3 se representaron los tiempos pedidos (como ciclos) para los quantums 2, 10 y 30 respectivamente.

pid	ciclos	inicio	fin	latencia	waiting time	tiempo total
0	71	0	373	2	302	373
1	71	0	377	7	306	377
2	71	0	381	12	310	381
3	13	0	81	17	68	81
4	13	0	84	20	71	84
promedio	-	-	-	11.6	TaskCPU: 306.0 TaskConsola: 69.5	TaskCPU: 377.0 TaskConsola: 82.5

Tabla 1: Tiempos para task_ej5.tsk con RR y quantum 2

pid	ciclos	inicio	fin	latencia	waiting time	tiempo total
0	71	0	221	2	150	221
1	71	0	232	35	161	232
2	71	0	243	68	172	243
3	13	0	252	101	240	252
4	13	0	255	104	242	255
promedio	-	-	-	62	TaskCPU: 161.0 TaskConsola: 241.0	TaskCPU: 232.0 TaskConsola: 253.5

Tabla 3: Tiempos para task_ej5.tsk con RR y quantum 30

pid	ciclos	inicio	fin	latencia	waiting time	tiempo total
0	71	0	177	2	106	177
1	71	0	180	21	109	180
2	71	0	265	28	194	265
3	13	0	272	41	259	272
4	13	0	279	44	266	279
promedio	-	-	-	27.2	TaskCPU: 136.3 TaskConsola: 262.5	TaskCPU: 207.3 TaskConsola: 275.5

Tabla 2: Tiempos para task_ej5.tsk con RR y quantum 10

Lo que podemos notar de las tablas, es que con quantums chicos, la latencia es menor, pero cuando hay muchas tareas de uso intenso de CPU, éstas tardan más en terminar debido a los constantes cambios de contexto. Para quantums más grandes, los proceso E/S parecen sufrir de inaninción, pues se ejecutan primero los proceso de CPU que son largos.

pid	ciclos	inicio	fin	latencia	waiting time	tiempo total
0	71	0	73	2	2	73
1	71	0	146	75	75	146
2	71	0	219	148	148	219
3	13	0	234	221	221	234
4	13	0	249	236	236	249
promedio	-	-	-	136.4	TaskCPU: 75.0 TaskConsola: 228.5	TaskCPU: 146 TaskConsola: 241.5

Tabla 4: Tiempos para tasks_ej5.tsk con FCFS

6. Ejercicio 6

6.1. Desarrollo

Ejecutamos el lote anterior con el scheduler FCFS y lo comparamos con el ejercicio anterior con un solo procesador.

En un scheduler FCFS (primero en llegar, primero en ser servido), el órden de prioridad de los proceso se manejan como una cola FIFO: se asigna el procesador, al proceso que llegue primero. En este algoritmo de scheduling, el proceso utiliza la CPU hasta completarse.

El algoritmo de scheduling de RR (Round Robin), es similar a FCFS en el sentido que maneja una cola FIFO, pero cada proceso tiene un período de tiempo que, una vez agotado, indica la scheduler desalojar la tarea y correr la siguiente, la tarea desalojada pasa al final de la cola en estado de "espera" hasta que vuelve a ser su turno para obtener el procesador. La gran diferencia entre un algoritmo FCFS y uno RR está justamente en éste último tiene un mecanismo de desalojo y el primero no.

6.2. Experimentación

Podemos ver que el gráfico de Gantt para el lote de tareas task_ej5.tsk de la figura 14 se asemeja más a la corrida del lote en `SchedRR` con quantum 30, que a las de quantums más chicos, pues las tareas intensas de CPU tienen un tiempo total de corrida (71 ciclos) casi el doble del quantum que se les asigna, y en RR, no son desalojadas las tareas hasta que no se les agota el quantum. Volcamos los datos de tiempo en la tabla 4 de tiempos para cada tarea

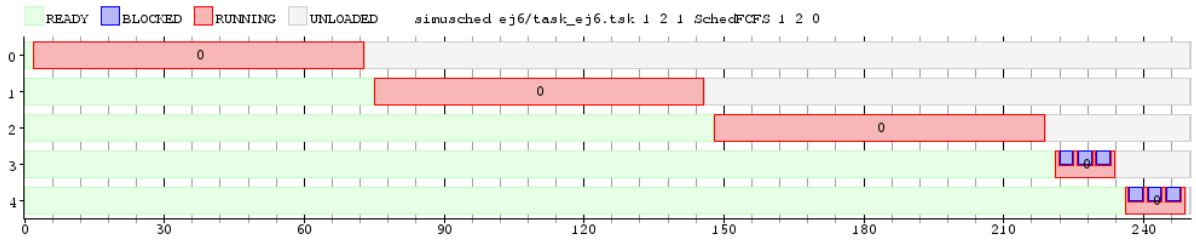

Figura 14: tasks_ej5.tsk con FCFS

FCFS y RR con quantum 30 tienen latencia promedio el doble una de la otra, y waiting times y tiempos totales similares para los preceso de E/S, pues en ambos casos, corren últimos y luego de que hayan corrido los proceso más largo de CPU. En cambio para las tareas intensas de CPU, éstos dos últimos valores son mejores en FCFS, justamente porque éstos procesos corren primeros y completos antes de cambiar a las tareas de E/S.

7. Ejercicio 7

7.1. Desarrollo

En éste ejercicio debíamos deducir que características tenía el *SchedMistery* y reproducir su código en *SchedNoMistery*. Para lo primero creamos varios lotes donde incluíamos muchas tareas del mismo tipo, y luego hicimos combinaciones de tipos de tareas.

Al completar ésta serie de tests pudimos deducir las características que enumeramos a continuación. Llamaremos E a los valores que se le pasa al scheduler.

1. *SchedMistery* es un scheduler de Multiples Colas con prioridad.
2. Siendo N la cantidad de entradas que se le da al programa (#E), el SchedMistery tendrá N+1 colas de prioridad, siendo la primera de 1 tick y el quantum del resto corresponde a los valores ingresados.
3. La cola de quantum 1 es la de mayor prioridad y el resto de colas va decendiendo su prioridad en el orden que se ingresan.

   Ej: Siendo prioridad N+1 la más alta

Cola N^{ro}	quantum	prioridad
1	1	N+1
2	E_1	N
3	E_2	N-1
4	E_3	N-2
5	E_4	N-3
i	E_{i-1}	N-i+2
N+1	E_N	1

4. Siempre se ejecuta la primer tarea en Ready de la cola no vacía de mayor prioridad.
5. Si una tarea corre el quantum completo (i.e. no se bloquea o termina) y hay una cola de prioridad menor, la tarea es eliminada de la cola E_i en la que se encuentra y pasa a la cola E_{i+1}.
   Si estaba en la cola de prioridad 1 (i.e. la última cola), se queda en ella.
6. Las tareas que corren el quantum completo en la última cola pasan al final de la cola, funcionando como un *RoundRobin* con el quantum de ésa cola.
7. Si una tarea es bloqueada, es eliminada de la cola en la que se encuentra y pasa a la cola E_{i-1} salvo que se encuentre en la primer cola. En cuyo caso se la vuelve a agregar al final de la misma.

7.1.1. Ejemplos y tests

Los siguientes son 3 lotes de tareas que ejemplifican las propiedades antes mencionadas. Los lotes reales que se usaron para la deducción son muchos y variados, y no aportan mayor información que los presentados a continuación:

Lote loteEj7a.tsk:

```
TaskCPU 30
TaskCPU 30
TaskCPU 30
TaskConsola 5 3 3
TaskConsola 5 3 3
```

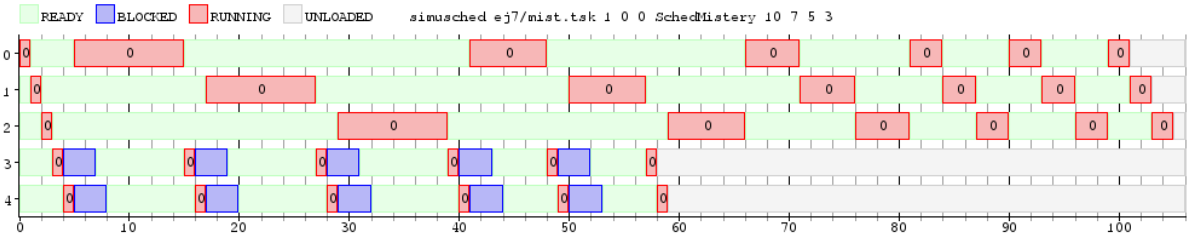

Figura 15: mist.tsk

En ésta figura se puede ver como el algoritmo hace ejecutar a las tareas TaskCPU primero 1 tick y luego la cantidad de ticks pasados como argumento. Finalmente, luego de hacer la primer corrida de 3 ticks vuelven a correr la misma cantidad como en un RR.

En las tareas TaskConsola sólo se observa que tienen mayor prioridad luego de la primer corrida, y que al estar bloqueadas dan paso a las tareas de menor prioridad.

Lote loteEj7b.tsk:

```
TaskCPU 50
TaskCPU 70
TaskBatch 70 3
@40:
TaskIO 40 25
TaskIO 15 35
@120:
TaskCPU 40
TaskCPU 35
```

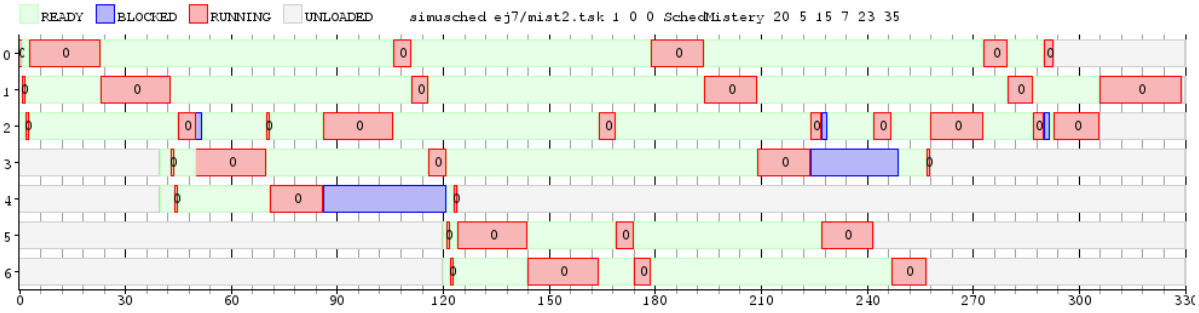

Figura 16: mist2.tsk

En la figura 16 se puede observar además de los comportamientos antes mencionados que las tareas bloqueadas pasan a la cola siguiente de mayor prioridad. Ésto se observa siguiendo las prioridades de las tareas bloqueadas, principalmente la tarea TaskBatch.

Lote loteEj7c.tsk:

```
TaskBatch 80 2
TaskBatch 80 2
TaskBatch 80 2
TaskBatch 80 2
```

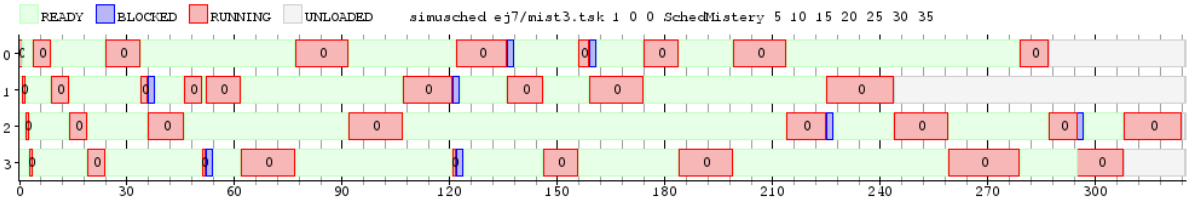

Figura 17: mist3.tsk

Para ejemplificar mejor lo antes dicho creamos este caso donde sólo se ejecutan TaskBatch. Con esto podemos ver los cambios de prioridad de las tareas.

7.2. Experimentación

Veremos que el *SchedNoMistery* se comporta igual que el *SchedMistery*:

La ejecución de *SchedNoMistery* para los ejemplos anteriores son efectivamente los mismos:

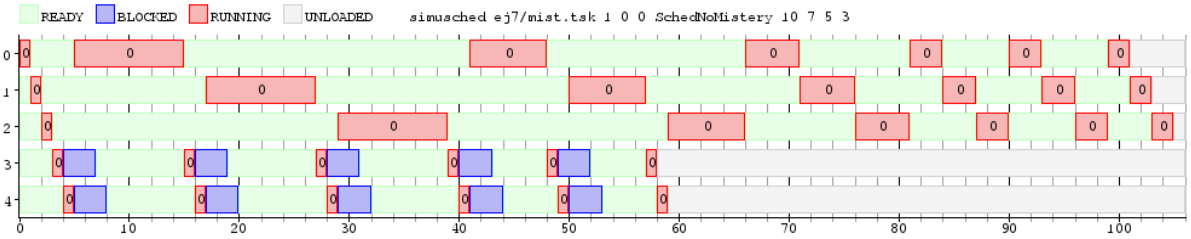

Figura 18: mist.tsk con SchedNoMistery

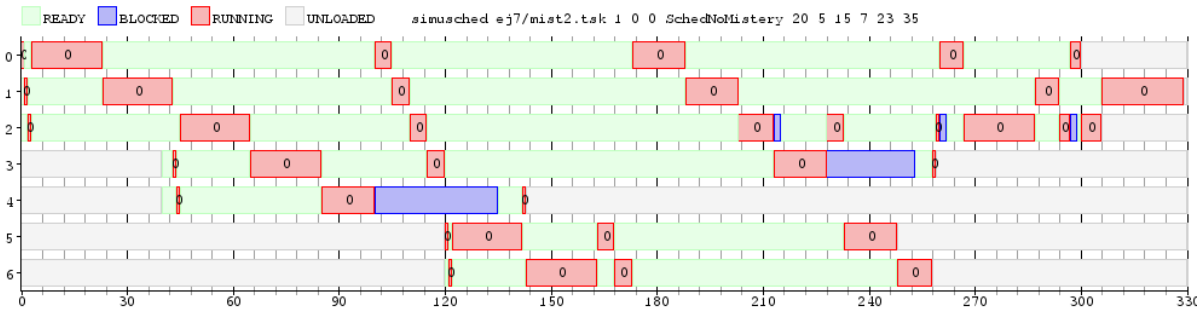

Figura 19: mist2.tsk con SchedNoMistery

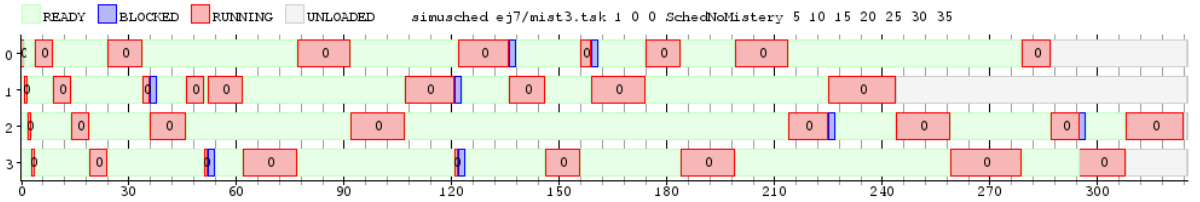

Figura 20: mist3.tsk con SchedNoMistery

8. Ejercicio 8

8.1. Desarrollo

La implementación del scheduler *Round-Robin* que no permite la migración de procesos entre núcleos es muy similar a la implementación original de *Round-Robin*. La diferencia es que en esta nueva implementación hay un vector de cola para los procesos. Esto es para poder indexar por el número de CPU y asi obtener la cola de procesos pendientes de ejecución.
En esta implementación tenemos un vector de listas de procesos bloqueados y un vector en el cual indicamos si el núcleo tiene un proceso ejecutándose o no. Nuevamente, elegimos un vector debido a que se puede indexar por el número de CPU y así obtener el dato correspondiente.
Entonces, el cardinal de cada una de estas estructuras va a ser igual a `RUNNING + BLOCKED + READY`. Se implementaron los mismos métodos que en la implementación original y se utilizó casi el mismo constructor, con la diferencia que ahora se genera una lista y una cola por cada cpu.
El método `load(pid)` fue el que mas cambios tuvo, ya que primero tenemos que sumar la cantidad de cada una de las colas, para luego sacar el mínimo y asignarlo al CPU correspondiente.
Una segunda modificación se llevo a cabo en el método `unblock(pid)` ya que debemos asignar el proceso desbloqueado a la CPU correspondiente y removerlo de la cola a la cual estaba asignada.

Los demás procesos conservan la misma implementación, con dos salvedades:

1. La primera, es el hecho de que en las listas son accedidas por un índice. Ese índice es la CPU que viene como parámetro.
2. La segunda, es el hecho de que nos interesa saber si el CPU esta ejecutando o no un proceso (distinto del IDLE). Debido a esto, al momento de retornar el próximo proceso a ejecutar el algoritmo marca una señal indicando si esta ejecutando IDLE o no.

8.2. Experimentación

Para las primeras pruebas ejecutamos el siguiente lote de tareas:

```
TaskCPU 50
TaskCPU 10
TaskCPU 50
TaskCPU 10
```

Con los siguientes parámetros de configuración:

Scheduler	Cantidad de núcleos	Cambio de contexto	Cambio de CPU	Quantum 1	Quantum 2
Round Robin	2	4	0	4	4
Round Robin 2	2	4	0	4	4

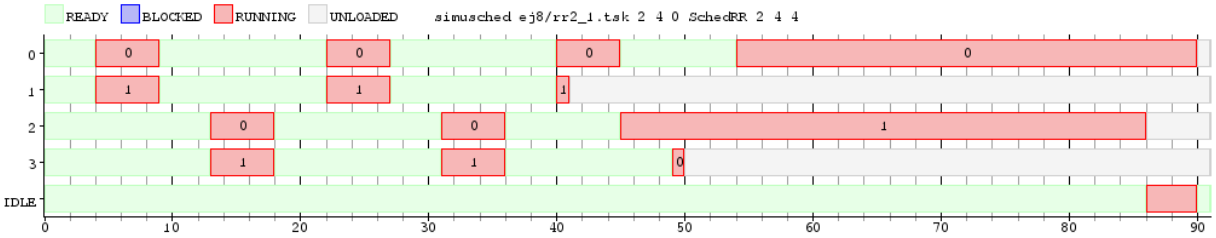

Figura 21: rr2Lote1.tsk

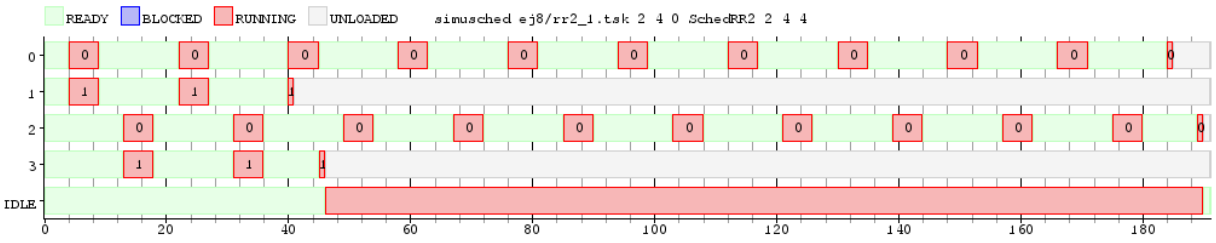

Figura 22: rr2Lote1.tsk

Para este lote lo que buscamos es que la segunda versión de round robin se encuentre en desventaja con respecto a la primera. Para ello, generamos un lote en donde hay cuatro tareas de las cuales dos se utiliza la CPU de manera intensiva.
Por la disposición en las cuales se encuentran ordenadas, en la segunda versión de round robin, vamos a tener un núcleo con dos tareas que utilizan mucha CPU con respecto a otro núcleo que tiene dos tareas livianas. Esto genera que en el segundo núcleo termine las dos tareas y quede disponible mientras el segundo sigue procesando.
Por el contrario, esta situación no se produce con nuestra primera versión del round robin, como la cola los dos CPU van tomando las tareas y las van procesando.
Entonces como consecuencia podemos ver a simple vista que la segunda versión del round robin tiene mas turnaround y desperdicia mas el CPU.

Para las próximas pruebas se ejecutó el siguiente lote de tareas:

```
TaskCPU 23
TaskCPU 7
TaskCPU 13
TaskCPU 11
```

Los parámetros de configuración fueron los siguientes:

Scheduler	Cantidad de núcleos	Cambio de contexto	Cambio de CPU	Quantum 1	Quantum 2
Round Robin	2	1	20	2	2
Round Robin 2	2	1	20	2	2

En esta simulación vamos a generar una situación en donde la segunda versión de round robin se encuentre en ventaja con respecto a la primera. En este caso no hacemos incapie en en el lote de tareas, sino en los parámetros. En este caso definimos un valor alto para la migración entre núcleos, ya que eso solamente se da en la primera versión de round robin, en la segunda no hay migración (ya que cada núcleo tiene su propia cola). Esto provoca que en la primera versión del round robin el turnaround sea mayor que la segunda versión.

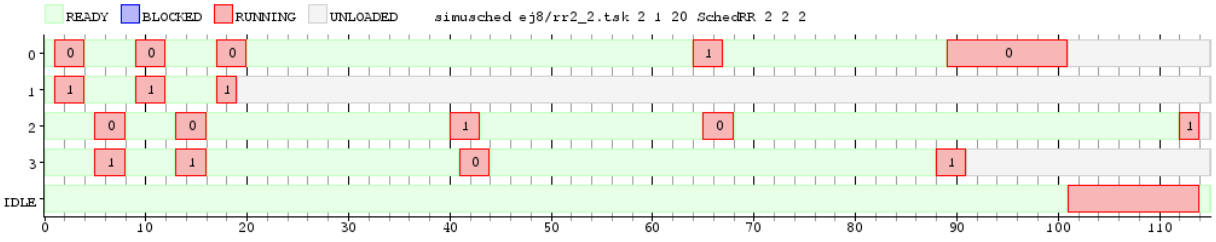

Figura 23: rr2Lote2.tsk

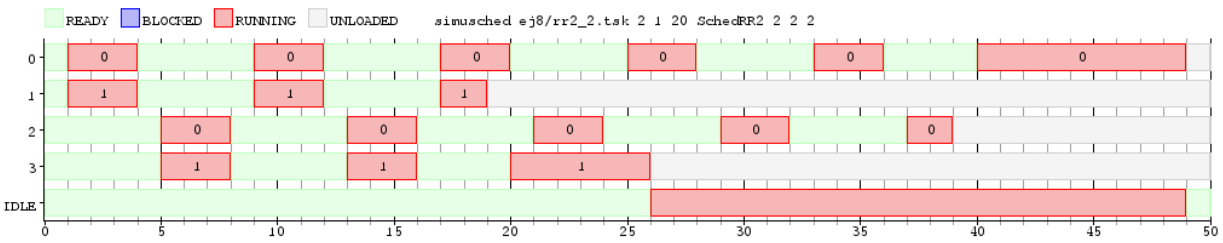

Figura 24: rr2Lote2.tsk