DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Sistemas operativos
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Federico De Rocco	403/13	fede.183@hotmail.com
Fernando Otero	424/11	fergabot@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 solamente se dedica a bloquearse una cantidad n de veces y con una duración al azar entre bmin y bmax, lo que hacemos para resolver este ejercicio es simplemente conseguirnos un número aleatorio n veces. Para esto usamos la función de c++ rand(), aclarando que queremos que los valores se encuentren entre los pedidos.

1.2. Experimentación

Para cumplir con lo pedido en el ejercicio utilizaremos el lote de tareas loteEj1.tsk, la representación del mismo usando la política FCFS es la siguiente:

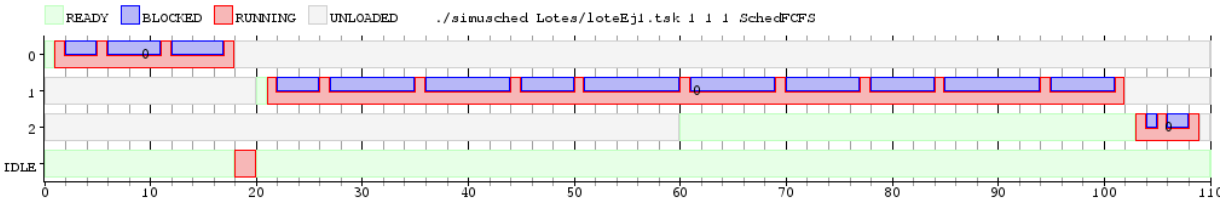

Figura 1: loteEj1.tsk con FCFS

Con este 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, podemos decir que el algoritmo es correcto.

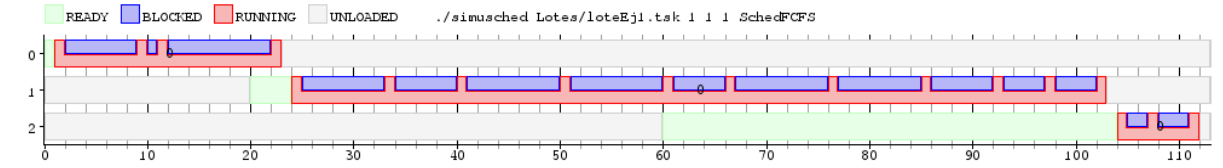

Figura 2: loteEj1.tsk con FCFS segundo intento

Ejecutamos, por segunda vez, en las mismas condiciones para mostrar como varían los tiempos de los bloqueos.

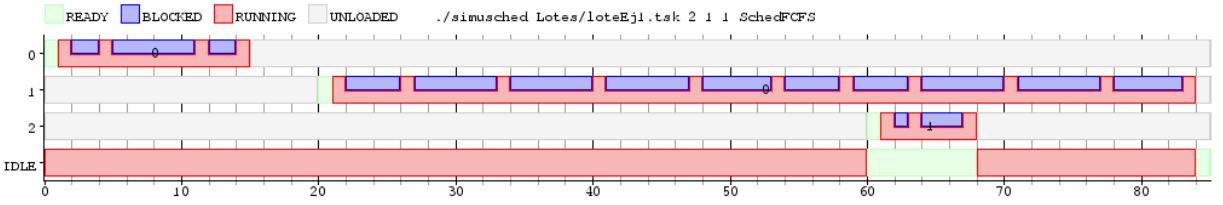

Figura 3: loteEj1.tsk con FCFS con dos cores

Por último lo hacemos con dos cores, para mostrar como se comporta.

2. Ejercicio 2

2.1. Experimentación

El loteEj2.tsk contiene lo pedido por enunciado.

TaskCPU 100 TaskConsola 20 2 4 TaskConsola 25 2 4

Veamos que tiene un uso de 100 de cpu que es del algoritmo complejo, un TaskConsola que es para la canción que realiza 20 llamadas bloqueantes y el otro de 25 para navegar por internet. Las llamadas bloqueantes son entre 2 y 4.

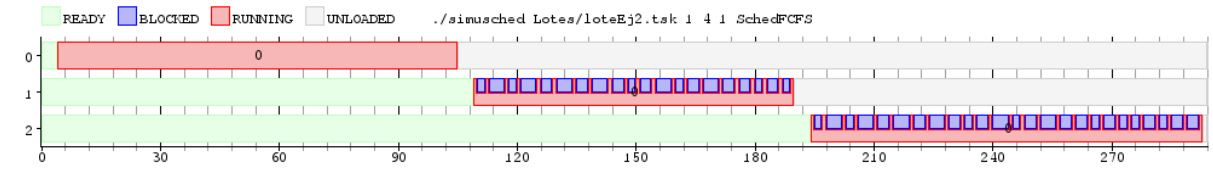

Figura 4: loteEj2.tsk con FCFS y coste cambio de contexto de 4 ciclos con 1 núcleo

Proceso	Latencia
CPU	4
Canción favorita	108
Navegar por Internet	194

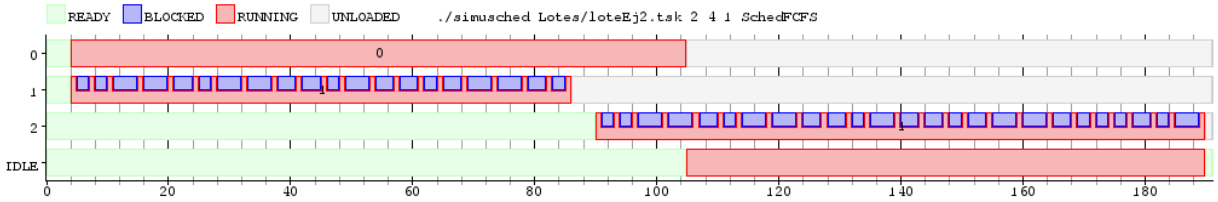

Figura 5: loteEj2.tsk con FCFS y coste cambio de contexto de 4 ciclos con 2 núcleo

Proceso	Latencia
CPU	4
Canción favorita	4
Navegar por Internet	91

Hay que aclarar que las latencias podrían variar ya que TaskConsola genera las llamadas bloqueantes con un tamaño aleatorio entre 2 y 4. Podemos ver que en el primer caso se debe esperar hasta que el algoritmo complejo termine para poder escuchar musica y , a su vez, que esto termine para poder navegar por internet. En el segundo caso, vemos que se puede ejecutar el algoritmo complejo y escuchar la canción al mismo tiempo, pero se debe terminar la canción para poder navegar por internet. Este último es mejor porque la canción y navegar por internet tienen latencias menores.

Rolando va a tener muchos problemas para ejecutar esto con un solo núcleo ya que mientras se este ejecutando el cpu no se podrá hacer ninguna de las otras cosas, que es lo que él pretende. Una buena solución sería tener 3 núcleos para que las tres cosas se puedan hacer al mismo tiempo. Otra sería usar Round-Robin para poder ir alternando los procesos y que no tenga que esperar que uno termine para poder ir ejecutando los otros.

3. Ejercicio 3

3.1. Desarrollo

Para poder resolver el problema de TaskBatch lo que hacemos es generar cant_bloqueos valores aleatorios no repetidos que no estén ubicados fuera de total_cpu. Lo generado por esto podría no estar ordenado, así que le hacemos un sort al resultado. Por último ciclaremos total_cpu veces y en las posiciones donde se tendría que bloquear hacemos la llamada usa_IO con duración de 1.

3.2. Experimentación

Siguiendo lo pedido por enunciado, realizaremos el gráfico del loteEj3.tsk usando el FCFS y nos queda lo siguiente:

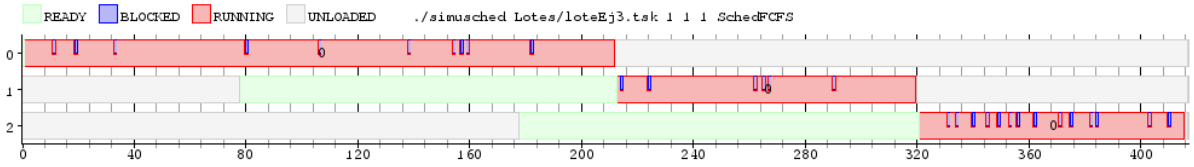

Figura 6: loteEj3.tsk con FCFS con un core

Podemos ver que se ejecutan pequeñas llamadas bloqueantes, estas tienen duración de un ciclo.

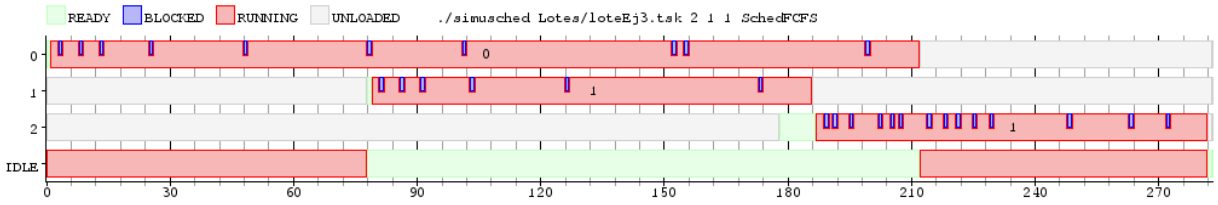

Figura 7: loteEj3.tsk con FCFS con dos cores

Mismo experimento con dos cores en lugar de uno.

4. Ejercicio 4

4.1. Desarrollo

El ejercicio consiste en programar un scheduler de Round-Robin, para esto utilizaremos las siguientes estructuras: vector de enteros pid_cores, vector de enteros pid_bloqueados, vector de enteros quantum_restantes, entero cant_cores, entero cpu_quantum, Una cola de enteros enEspera.

Cuando creamos el scheduler se nos dan la cantidad de cores(que será cant_cores) y el quantum de los cpus(cpu_quantum). Según el enunciado los procesos están agrupados en una única cola, de ahí viene enEspera. Como conocemos la cantidad de cores usamos tres estructuras de vectores para, solo teniendo el número del cpu, poder acceder a la información de las mismas. pid_bloqueados nos indica si el proceso en cuestión está o no bloqueado(Estos pids no están en enEspera ya que esta corresponde a los ready), pid_cores nos da el pid del proceso que esta corriendo en el cpu y quantum_restantes nos dice cuanto tiempo le queda por ejecutar hasta terminar su tiempo en el procesador.

En el programa load en el cual tenemos que poner un proceso en el scheduler lo que haremos será ponerlo en enEspera. En el caso de unblock lo que hacemos es eliminar el pid de pid_bloqueados, sabemos que está ahí porque lo añadimos en tick. Por último tenemos el programa tick que se encarga de realizar los procedimientos pertinentes en cada tick de reloj. Tenemos tres casos de motivos:

TICK: en el mismo sabemos que ha pasado un tick de reloj y debemos disminuir los quantum que le quedan al procedimiento del cpu. En el caso de que estos terminen en 0 debemos plantearnos si debemos desalojar la tarea para poner otra(Si es la IDLE no hacemos nada). De ser el caso, pondremos la tarea actual en la cola enEspera y el próximo elemento en esta será el nuevo que se hirá ejecutando en este núcleo. Reseteamos el quantum_restantes para que tenga el valor de cpu_quantum. Notece que si no hay más tareas excepto por la que se estaba ejecutando se volverá a ejecutar la misma.

BLOCK: Si el proceso se va a bloquear añadimos el pid a pid_bloqueados, si existe algún proceso en ready, lo ponemos a ejecutar en ese core. De solo ser un tick en el cual está bloqueado, solo hacemos lo último(sin añadir a pid_bloqueados).

EXIT: Colocamos en el core la tarea IDLE y si la cola no está vacía, le asignamos la tarea que se encuentre próxima en ella. Después reseteamos los quantum del proceso.

5. Ejercicio 5

5.1. Desarrollo

En el ejercicio se pide diseñar un lote con tres tareas de 50 ciclos y dos de 5 llamadas bloqueantes de 3 ciclos de duración. El lote que representamos inicia todas las tareas a la vez

```
TaskCPU 50
TaskCPU 50
TaskCPU 50
TaskConsola 5 3 3
TaskConsola 5 3 3
```

Ejecutamos el lote con el scheduler `SchedRR` para quantums 2, 10 y 50 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 50, el valor correspondiente es 51 (50 ciclos de uso de CPU más 1 ciclo para EXIT)
Para `TaskConsola` de parámetros 5 3 3, son 21 ciclos, pues son 5 ciclos de E/S de 3 ciclos de duración más 1 ciclo de CPU cada E/S más un ciclo para EXIT (15 + 5 +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

5.2. Experimentación

En las figuras 8, 9 y 10 se representaron los diagramas de Gantt para los quantums 2, 10 y 50 respectivamente. Se puede ver, cómo a quantums más chicos (figura 8), 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 8: loteEj5.tsk con RR y quantum 2

Para quantums más grandes, todas las tareas terminan muy próximas entre sí, pero a quantums de 50 (figura 10), las tareas de E/S comienzan luego de 150 ciclos (los tres quantums de las tareas de CPU) y podría parecer que el SO no responde.

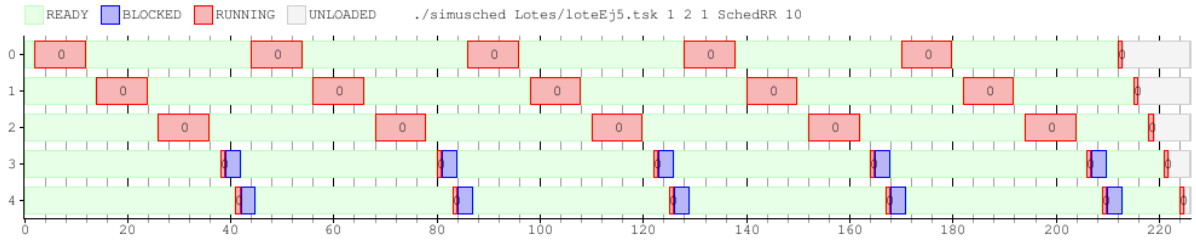

Figura 9: loteEj5.tsk con RR y quantum 10

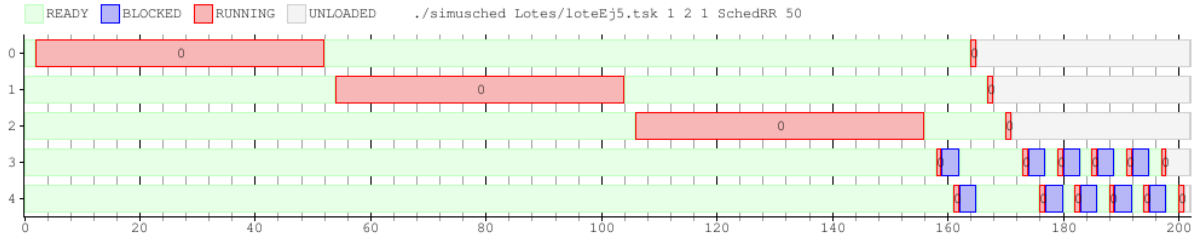

Figura 10: loteEj5.tsk con RR y quantum 50

En las tablas 1, 2 y 3 se representaron los tiempos pedidos (como ciclos) para los quantums 2, 10 y 50 respectivamente.

pid	ciclos	inicio	fin	latencia	waiting time	tiempo total
0	51	0	339	2	288	339
1	51	0	342	6	291	342
2	51	0	345	10	294	345
3	21	0	105	14	84	105
4	21	0	108	17	87	108
promedio	-	-	-	9.8	CPU: 291 E/S: 85.5	CPU: 342 E/S: 106.5

Tabla 1: Tiempos para loteEj5.tsk con RR y quantum 2

pid	ciclos	inicio	fin	latencia	waiting time	tiempo total
0	51	0	165	2	114	165
1	51	0	168	54	117	168
2	51	0	171	106	120	171
3	21	0	198	158	177	198
4	21	0	201	161	180	201
promedio	-	-	-	96.2	CPU: 117 E/S: 178.5	CPU: 168 E/S: 199.5

Tabla 3: Tiempos para loteEj5.tsk con RR y quantum 50

pid	ciclos	inicio	fin	latencia	waiting time	tiempo total
0	51	0	213	2	162	213
1	51	0	216	14	165	216
2	51	0	219	26	168	219
3	21	0	222	38	201	222
4	21	0	225	41	204	225
promedio	-	-	-	24.2	CPU: 165 E/S: 202.5	CPU: 216 E/S: 223.5

Tabla 2: Tiempos para loteEj5.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	51	0	53	2	2	53
1	51	0	106	55	55	106
2	51	0	159	108	108	159
3	21	0	182	161	161	182
4	21	0	205	184	184	205
promedio	-	-	-	102	CPU: 55 E/S: 172.5	CPU: 106 E/S: 193

Tabla 4: Tiempos para loteEj5.tsk con FCFS

6. Ejercicio 6

6.1. Desarrollo

En este ejercicio se pide ejecutar el lote anterior con el scheduler FCFS y compararlo con el ejercicio anterior con un solo procesador.

6.2. Experimentación

El gráfico de Gantt para el lote de tareas loteEj5.tsk se exhibe en la figura 11.

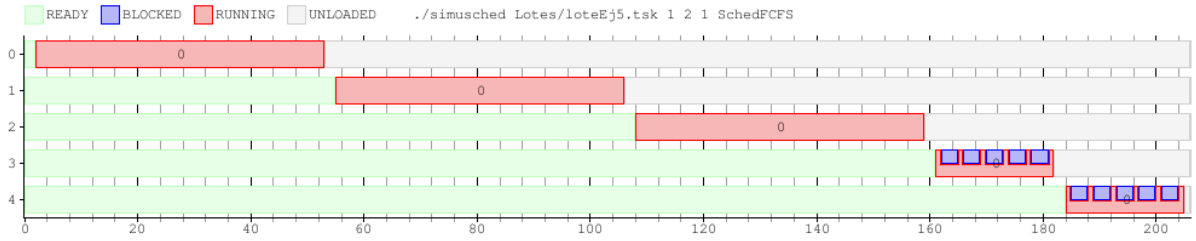

Figura 11: loteEj5.tsk con FCFS

Podemos ver que el gráfico se asemeja mucho a la corrida del lote en `SchedRR` con quantum 50. Volcamos los datos de tiempo en la tabla 4 de tiempos para cada tarea

FCFS y RR con quantum 50 tienen latencia promedio similares, y waiting times y tiempos totales similares también 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

Analizamos con varios grupos de tareas el comportamiento de *SchedMistery* para llegar a los siguientes resultados:

(a) El scheduler *SchedMistery* es un RoundRobin modificado

(b) La modificación se da en el tiempo que se le permite correr a cada tarea antes de ser desalojada

(c) Dicha modificación tiene en cuenta dos cosas: los parámetros de entrada del programa y si quedan otras tareas que pueden correr.

(d) Si no hay otras tareas que puedan correr no hay desalojo, la tarea Idle sólo aparece cuando no hay tareas que puedan correr.

(e) En cuanto a los tiempos de ejecución de cada tarea, la primera aparición de la tarea es de un tick de reloj, mientras que el resto equivale al valor de la entrada, tomada secuencialmente (explicado más abajo con los gráficos)

(f) Las ejecuciones de las tareas fuera de los tiempos dados en la entrada del programa son iguales a la última corrida correspondiente a los valores de la entrada.

(g) De no haber otras tareas que puedan ejecutarse, se siguen contando los valores de los parámetros de entrada

7.2. Experimentación

Ahora mostraremos de donde obtuvimos la información antes mencionada

Los lotes utilizados fueron:

loteEj7a.tsk
```
TaskCPU 18
TaskCPU 18
TaskCPU 18
TaskCPU 18
```

loteEj7a.tsk
```
#Tarea intensiva de CPU 1
TaskCPU 18
#Tarea intensiva de CPU 2
@7:
TaskCPU 20
#Tarea interactiva 1
TaskConsola 4 3 6
#Tarea interactiva 2
TaskConsola 8 2 7
```

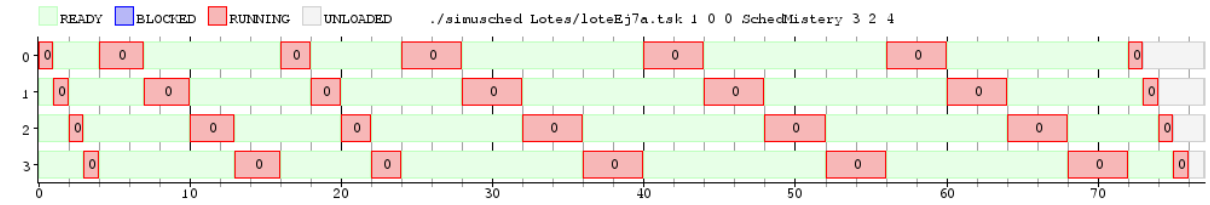

Figura 12: Se pueden apreciar (a)(b)(e)(f)

En esta figura se puede ver claramente que es un Round Robin y que los tiempos que toman las tareas cada vez que entran en Ready son 1, 3, 2, 4, 4.. Luego de la cuarta vez que el scheduller pone a correr la tarea, siempre le da 4 tics de reloj

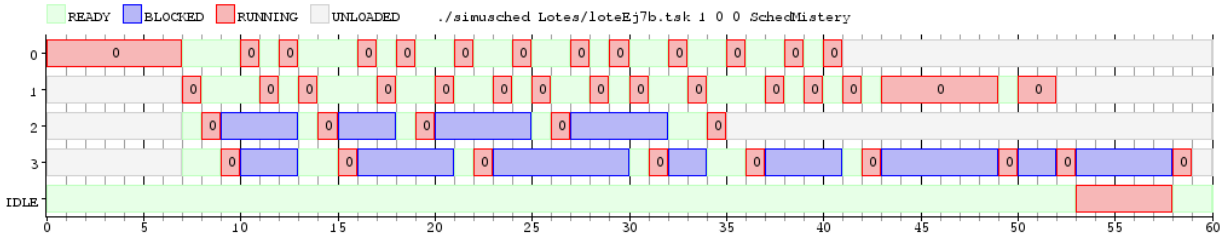

Figura 13: Se pueden apreciar (a)(b)(d)

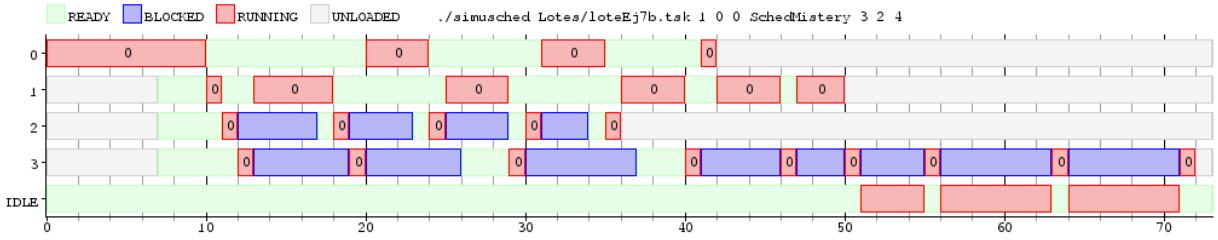

Figura 14: Se pueden apreciar (a)(b)(c)(e)(f)(g)

8. Ejercicio 8

8.1. Desarrollo

Similar a lo visto en el ejercicio 8, tendremos que programar la estructura de un scheduler de Round-Robin. La diferencia es que en este caso en vez de tener una cola única que vaya guardando todos los procesos, cada core tendrá su propia cola de procesos. Para resolver esto utilizaremos una estructura para representar el core que nos dará la siguiente información: un entero pidActual, un entero quantum_restantes, su cola de pids enEspera, un vector pid_bloqueados que indica cuales están en ese estado.

Después tenemos la cantidad de cores y el quantum de los mismos. Por último necesitamos agrupar las estructuras core que tenemos. Para esto usamos simplemente un vector (nucleos).

La función load lo que hace es buscar cual de los cores tiene menor cantidad de procesos(suma los bloqueados, los de enEspera y el de pidActual, si no es IDLE). Una vez encontrado, lo encola en el enEspera del core.

unblock lo que hace es buscar cual es el core donde se encuentra el pid que se desea bloquear, una vez encontrado lo quita de pid_bloqueados y lo pone en enEspera.

Por último tenemos la función tick. El código es equivalente al del ejercicio 4.

8.2. Experimentación

Según lo pedido debemos mencionar un caso donde la migración de núcleos sea beneficiosa y otro donde no. Para lo segundo, usaremos dos core y cuatro tareas que irán apareciendo en este orden: la más costosa, la menos costosa, la segunda más costosa y la segunda menos costosa. Usaremos, en los dos cores, un quantum de 20 y nos queda lo siguiente:

Figura 15: loteEj8Experimento1.tsk con RR2

Para el caso en el que sea beneficiosa vamos a usar el mismo lote de tareas, pero las cambiaremos de orden para que sean así: la más costosa, la segunda más costosa, la menos costosa y la segunda menos costosa. Nos queda lo siguiente:

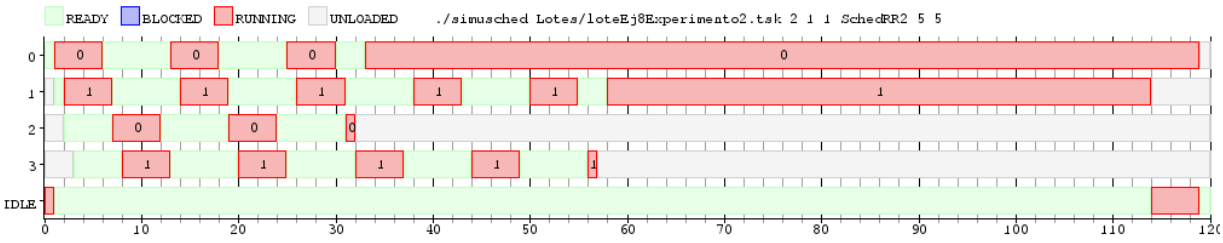

Figura 16: loteEj8Experimento2.tsk con RR2

Podemos ver que efectivamente el primer gráfico muestra que se tarda más en completar los procesos. Esto se debe a que el orden que le dimos a las tareas hace que el primer procesador se le asigne las tareas más largas y costosas, mientras que el segundo tiene las más cortas. En ese sentido se muestra como la mayor parte del trabajo recae en uno de ellos mientras que el otro termina y queda en ready mucho más rápido. Para el segundo experimento, al cambiar el orden podemos apreciar que los dos procesadores tienen una tarea muy costosa y otra no tanto. Efectivamente termina más rápido ya que el trabajo se distribuye de forma más justa.

9. Referencias

Referencias