DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Scheduling

Sistemas Operativos

Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Arístides Catalano	279/10	dilbert_cata@hotmail.com
Laura Muiño	399/11	mmuino@dc.uba.ar
Jorge Quintana	344/11	jorge.quintana.81@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

Contents

1 Ejercicio 1

El ejercicio consiste en programar una tarea que simule n llamadas bloqueantes (con n pasado por parámetro) donde la duración de cada llamada bloqueante está determinada por los parámetros bmin y bmax.

Para generar las duraciones al azar dentro del rango [bmin, bmax], se utilizó la funcion rand de la libreria stdlib y se realizó lo siguiente:

int cant = bmax - bmin + 1;
int cant_ciclos_bloqueada;
cant_ciclos_bloqueada = bmin + (rand() % cant);

De esta forma se garantiza que *cant_ciclos_bloqueada* esté entre el rango pedido. Para realizar la simulación de las llamadas bloqueantes se ultilizó la función uso_IO provista por la cátedra.

Para probar la tarea se crearon los siguientes lotes:

- TaskConsola 5 10 20
- TaskConsola 15 2 20

En la primera tarea se eligieron esos parámetros para comprobar que la cantidad de tiempo bloqueado respetara los límites requeridos, dado que la primera simulación no permitía apreciar gráficamente a simple vista la variabilidad dentro del rango, se creo la segunda tarea, con un rango más amplio para que el diagrama de Gantt manifieste dicha variabilidad en los tiempos de bloqueo random de la tarea.

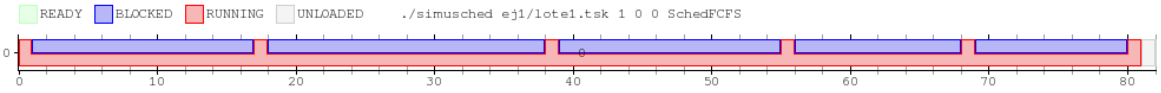


Figure 1: Lote con 1 core, sin context switch. Sched.: FCFS

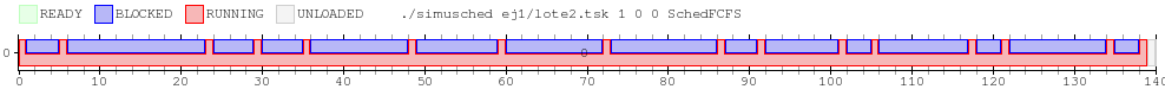


Figure 2: Lote con 1 core, sin context switch. Sched.: FCFS

2 Ejercicio 2

Para representar el uso de 100 ciclos de CPU y dos tareas bloqueantes definimos el siguiente lote de tareas:

- TaskCPU 100
- TaskConsola 20 2 4
- TaskConsola 25 2 4

TaskConsola realizará 20 y 25 llamadas bloqueantes respectivamente, con cada bloqueo de duración variable entre 2 y 4 ciclos. El cambio de contexto se fijo a 4 ciclos. En este ejercicio, el enunciado no aclaraba nada de costos de migración, con lo cual lo mantuvimos en cero. Estos fueron los gráficos obtenidos :

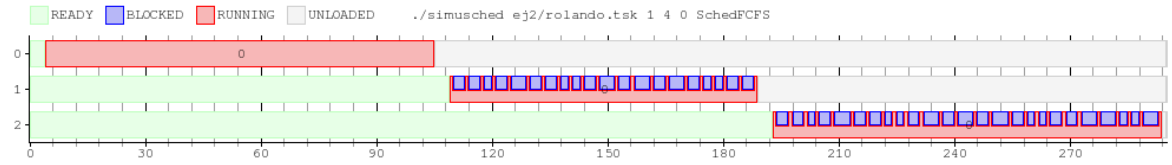


Figure 3: Un núcleo.

La latencia es el tiempo que un proceso tarda en empezar a ejecutarse. En la figura 3, con un núcleo, la latencia para las tareas será la suma entre el costo de cambio de contexto, la duración de la tarea anterior y la latencia de la tarea anterior. En el caso de la primer tarea (la cero) su latencia es 4, ya que no tiene tareas anteriores. La segunda tarea (la uno) tendrá latencia $4 + 100 + 4 = 108$. Por último, la tarea 2 tendra latencia $108 + duracion\_tarea_1 + 4$. La duración de la $tarea_1$ viendo el gráfico es aproximadamente de 78 ciclos, con lo que su latencia es 190 ciclos.

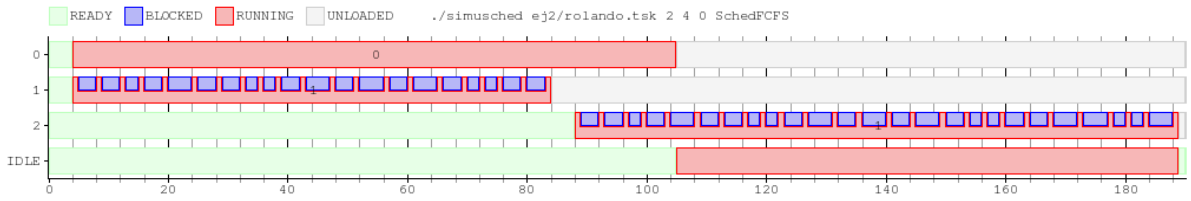


Figure 4:

En cambio, en la figura 4 con dos núcleos, las tareas cero y uno tiene latencia 4, ya que ambas corren en núcleos distintos. Luego, la tarea dos tienen latencia $4 + duracion\_tarea_1 + 4$. La duración de la $tarea_1$ es de 80 ciclos, con lo que la latencia de la tarea dos es 88.

Dados los valores de latencia de cada tarea en ambos gráficos, concluimos que la desventaja de tener un solo núcleo es el aumento de la latencia. Todas las tareas deben esperar a que el CPU esté libre para correr, con lo cual el valor de latencia de una tarea refleja no solo su context switch, sino que además, el tiempo de corrida y la latencia de las tareas anteriores.

3 Ejercicio 3

Para realizar la tarea TaskBatch, se creó un arreglo de tamaño $total_cpu$, donde en cada posición se guarda un valor booleano que indica si en dicho momento se genera una llamada bloqueante o no. Luego se generaron cant_bloqueos números distintos de manera aleatoria dentro del rango [0, total_cpu -1]. Al hacerlo se tuvo en cuenta el hecho de que se pueden obtener números repetidos. En este caso, si el momento creado coincide con otro momento ya generado, se vuelve a generar otro número hasta obtenerlos a todos distintos. Para cada "momento" generado, se indica en el arreglo anteriormente nombrado, el valor de verdad true (o 1). Finalmente, se recorre el arreglo procediendo a llamar a la función que simula los bloqueos (uso_IO) en caso de que la posicion del arreglo indique true, o a la función que simula el uso del CPU (uso_CPU) en caso contrario.

Se realizó una simulación para TaskBatch con el siguente lote de tareas:

- TaskBatch 10 3
- TaskBatch 5 2
- TaskBatch 8 4

El diagrama de Gantt obtenido es el siguiente:

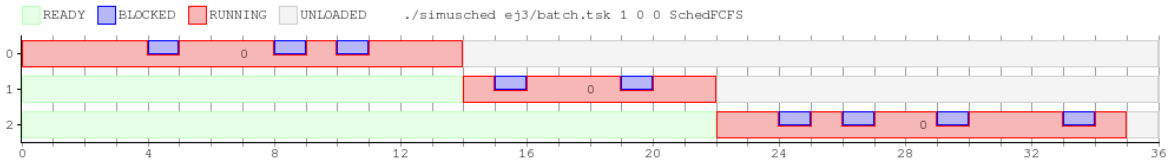


Figure 5: Primer lote para TaskBatch.

En la figura 5 se puede observar que para cada tarea, se realizan la cantidad de bloqueos pedidias. Por otra parte la duración de cada tarea equivale a la duración inicial más un cilo, con lo que concluimos que ese último ciclo correspondiente al exit() realizado por el scheduler.

Decidimos realizar otro lote, para verificar que la tarea funcionara adecuadamente.

- *3 TaskBatch 40 7

Se eligió el scheduler FCFS y un número de cores igual a la cantidad de tareas en el lote con el fin de que esté ejecutando de manera simultánea las 3 tareas y así ver efectivamente que las llamadas bloqueantes ocurren en momentos distintos para cada tarea. El diagrama de Gantt obtenido es el siguiente:

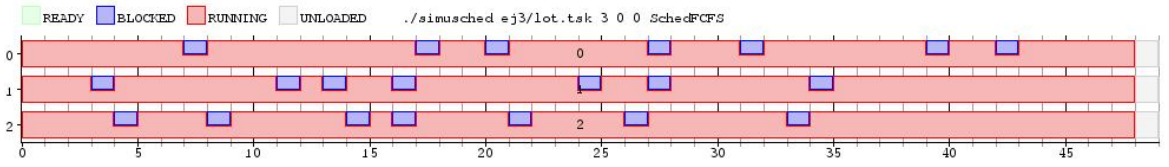


Figure 6: Tareas Batch

Se puede ver fácilmente como las tres tareas hacen los llamados en momentos distintos.

4 Ejercicio 4

El ejercicio consiste en implementar el scheduler $Round-Robin$, que soporte migraciones entre núcleos y tenga una única cola global para los procesos.

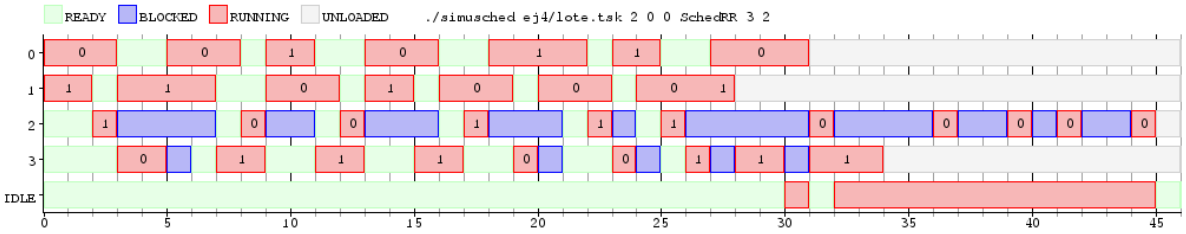
Para esto se utiliza la estructura $queue < int >$, donde cada elemento de la misma representa un pid de algún proceso listo para ser ejecutado. También se debe guardar la cantidad de cores y sus respectivos quantums, que son pasados como parámetro cuando se llama al constructor de la clase. Se almacenan entonces estos datos y el quantum que le queda a cada proceso en cada CPU, usando dos estructuras de tipo $vector < int >$.

Los métodos a implementar son:
$load(pid)$: encola pid en $colaReady$ que es la cola de procesos en estado ready. $unblock(pid)$: hace la misma acción, encolar nuevamente el pid de la tarea que se desbloqueó. Es decir, la tarea queda al final de la $colaReady$ En el método $tick(cpu, motivo)$ se toman 3 casos: el proceso consumió todo el ciclo usando el CPU, realizó una llamada bloqueante o terminó de ejecutarse.

- TICK: Si la tarea era IDLE_TASK y no hay procesos en $colaReady$ sigue ejecutando lo mismo. Si la tarea era IDLE_TASK pero hay tareas disponibles en $colaReady$ se toma el tope de la cola (se quita de la cola), se actualiza el quantum disponibles y se devuelve ese pid. Si era cualquier otra tarea se resta un tick al quantum actual. Se chequea si se agotó su quantum, sino sigue ejecutando. Si lo consumió todo, se encola el pid en colaReady, se actualiza el quantum diponible, y se toma el tope de la cola como pid siguiente.
- BLOCK: Si no hay mas tareas en colaReady, se devuelve IDLE_TASK. Si hay, se actualiza el quantum diponible se devuelve el tope de colaReady quitando el elemento de la misma.
- EXIT: Ídem BLOCK.

Se creó un lote de tareas con el fin de probar el correcto funcionamiento del scheduler.

- *2 TaskCPU 20
- TaskConsola 10 1 5
- TaskBatch 15 5



En el gráfico se puede ver, como las tareas van migrando de núcleo, y que cada núcleo tiene un quantum asociado. En particular el core 0 tiene quantum 3 y el core 1 tiene como quantum 2. Cada vez que cada tarea agota su tiempo es desalojada y encolada en $colaReady$. Cuando las tareas 2 y 3 realizan llamadas bloqueantes, son desalojadas y quitadas de la $colaReady$, una vez que se desbloquean se encolan nuevamente en $colaReady$ En el instante 5 se ve que como las tareas 2 y 3 estan bloqueadas la tarea 1 obtiene el cpu dos veces seguidas, ya que no queda ninguna otra tarea para ejecutar.

5 Ejercicio 5

Implementado el Scheduler Round-Robin, se desea analizar ciertos parámetros de calidad del scheduler comparando los mismos para un quantum de 2, 10 y 50 ciclos. El lote de tareas para la prueba consiste en el siguiente:

```
*3 TaskCPU 50
*2 TaskConsola 5 3 3
```

Que especifica tres tareas con uso de CPU durante 50 ciclos y 2 tareas que realizan 5 llamadas bloqueantes con duracion de 3 ciclos cada una.

Los parámetros de calidad a analizar son: latencia, waiting time y tiempo total de ejecución, siendo la latencia el tiempo que la tarea pasa en estado "listo" hasta que comienza su ejecución, el waiting time el tiempo que pasa una tarea sin ejecutar desde el momento que está en estado "listo" hasta que termina su ejecución, y el tiempo total de ejecución el tiempo que toma la tarea desde que está en estado "listo" hasta que cumple con su ejecución.

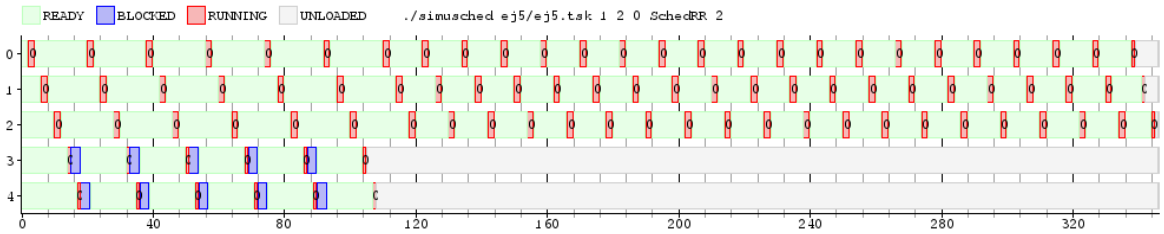


Figure 7: Tareas con quantum 2.

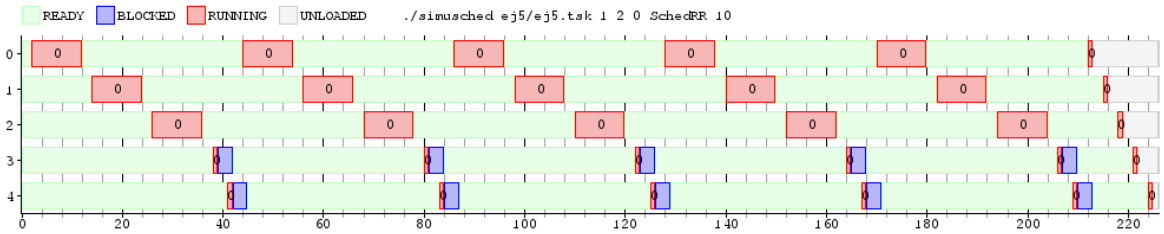


Figure 8: Tareas con quantum 10.

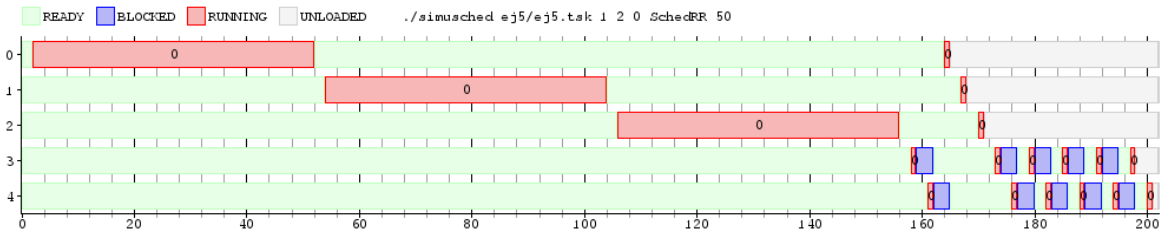


Figure 9: Tareas con quantum 50.

Como puede verse en la tabla 1, la latencia es menor cuando el quantum es 2 y va subiendo a medida que sube el quantum, esto sucede dado que el scheduler round robin cuando agota su quantum desaloja a

Table 1: Parámetros de calidad para el sceduler round robin: SchedRR.

	Tarea	Latencia	Waiting time	Total ejecución	Ratio (Espera/total)
Quantum = 2	0	2	288	339	0.85
	1	6	291	342	0.85
	2	10	294	345	0.85
	3	14	99	105	0.94
	4	17	102	108	0.94
	Avg	9.8	214.8	227.8	0.89
Quantum = 10	0	2	162	213	0.76
	1	14	165	216	0.76
	2	26	168	219	0.76
	3	38	216	222	0.97
	4	41	219	225	0.97
	Avg	24.2	186	219	0.84
Quantum = 50	0	2	114	165	0.69
	1	54	117	168	0.69
	2	106	120	171	0.70
	3	158	192	198	0.97
	4	161	195	201	0.97
	Avg	96.2	147.6	180.6	0.80

la tarea en ejecución y toma una nueva dentro de la cola de tareas que tiene definida. A menor quantum más se apega el scheduler al concepto de time-sharing mientras que a mayor quantum más tienen que esperar las tareas encoladas para empezar su ejecución.

En cuanto a waiting time y tiempo total de ejecución (que están íntimamente relacionados dado que el waiting time es la diferencia entre el tiempo total de ejecución y el tiempo que cada tarea posee el uso de los recursos de CPU), ambos parámetros son menores en promedio mientras mayor es el quantum otorgado a cada tarea. Esto sucede debido a que hay menor cantidad de cambios de contexto mientras más grande es el quantum otorgado a cada tarea. Sin embargo el promedio no es totalmente significativo dado que depende fuertemente del tipo y naturaleza de tareas que se ejecutan. Puede verse, por ejemplo, que si bien las tareas que hacen consumo intensivo de CPU mejoran sensiblemente sus parámetros mientras que las tareas que utilizan llamadas bloqueantes se ven afectadas negativamente.

Si tomamos como parámetro de "justicia" de un scheduler que sean favorecidos los procesos que menor cantidad de CPU toman, podemos concluir que es más justo un quantum menor, aunque debido al costo de cambio de contexto el cálculo a efectuar debe ser cuidadoso para no desperdiciar demasiado tiempo en el mismo. Se debe tener en cuenta que los valores tomados son basados en lotes de tareas conocidos y en un sistema interactivo hay mayor variabilidad en la naturaleza de las mismas.

6 Ejercicio 6

Con el análisis efectuado para el Scheduler Round Robin, utilizamos ahora el mismo lote de tareas para ejecutar una simulación con el Scheduler FCFS y comparar ambos. El lote de tareas para la prueba consiste en el siguiente:

```
*3 TaskCPU 50
*2 TaskConsola 5 3 3
```

La simulación arroja la siguiente salida (en formato gráfico)

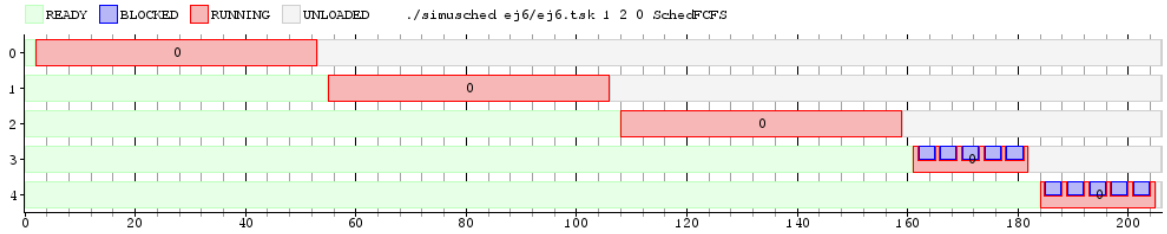


Figure 10: Tareas para Scheduler FCFS.

Analizando el gráfico y la salida análitica, llegamos a los siguientes valores:

Table 2: Parámetros de calidad para el sceduler FCFS: SchedFCFS.

	Tarea	Latencia	Waiting time	Total ejecución	Ratio (Espera/total)
FCFS	0	2	2	53	0.038
	1	55	55	106	0.519
	2	108	108	159	0.679
	3	161	176	182	0.967
	4	184	199	205	0.971
	Avg	102	108	141	0.635

Puede verse de estos valores que, por un lado, en los procesos que tienen sólo uso de CPU el "waiting time" es igual a la latencia, mientras que en aquellos que hacen utilización de recursos de entrada/salida el "waiting time" es igual a la latencia más el tiempo que la tarea permanece bloqueada.

Comparando las tablas 1 y 2 puede verse que en promedio la latencia empeora utilizando Scheduler FCFS en comparación con el Scheduler Round-Robin.

Respecto al "waiting time", el mismo mejora en promedio con respecto a todos los quanta probados con el Scheduler Round Robin, excepto para el caso de tareas con utilización de recursos de entrada/salida cuando el quantum es igual a 2.

La diferencia entre un Scheduler Round Robin y un Scheduler FCFS es más notoria respecto a las tareas con utilización de recursos de entrada/salida, dado que cuando no se realizan llamadas bloqueantes un Scheduler FCFS es un Scheduler Round Robin llevado al extremo, esto es, con un quantum lo suficientemente grande para que se pueda ejecutar cada tarea dentro del quantum asignado.

Cuando se tienen en cuenta las tareas que hacen entrada/salida, puede verse que en realidad FCFS desfavorece dichos procesos pues desperdicia ciclos de CPU esperando el desbloqueo, mientras que un Scheduler Round Robin, aprovecha para cambiar el contexto y otorgar CPU a otra tarea, incluso el tiempo de cambio de contexto se superpone al tiempo de bloqueo (esto puede verse claramente en las figuras 8 y 9 sobre las tareas 3 y 4. Pensando en un extremo en el que la entrada/salida bloquea al CPU durante muchos ciclos, puede concluirse que el Scheduler que más favorece este tipo de tareas es el Scheduler Round Robin con quantum pequeño.

7 Ejercicio 7

En este ejercicio se pide estudiar el comportamiento del scheduler Mistery y realizar una implementación que lo imite. Para esto se realizaron una serie de simulaciones variando primero la cantidad de parámetros y luego usando distintos lotes de tareas.

Las principales características observadas del scheduler son:

- La cantidad de parametros recibidos indican la cantidad de *colasReady* que va a tener el Sched.
- Siempre hay una cola con quantum 1, es la primera.
- Cada parámetro es el quantum asignado a cada cola. Si los argumentos son 263, la primera cola tendrá 2 ciclos de clock la segunda 6 y la tercera 3.
- El método *load*(*pid*) encola a las tareas en la primera colaReady.
- Si una tarea termina el quantum asignado a esa cola, se le quita el cpu y se la encola en la cola siguiente. Si estaba en la ultima cola, se la encola ahí.
- Si se le pasa un 0 como parámetro una vez que llegue una tarea a esa cola, va a ejecutarse hasta hacer EXIT.
- Las colas tienen prioridad, y van de menor a mayor. Es decir, si una tarea esta en ejecutando en la cola *colaReady*[2] y llega tareaNueva, se va a encolar en *colaReady*[0] y una vez que tarea sea desalojada, se le asignará el cpu a tareaNueva, ya que su cola tiene mayor prioridad.
- Si una tarea se bloquea se desaloja y se encola en la *colaReady* anterior a la que estaba cuando hizo la llamada bloqueante. Si estaba en la primera cola, se encola en la primera de vuelta.

Se crearon 3 lotes de tareas que permitan identificar fácilmente estas características en los diagramas de Gantt.

el primer lote de tareas es:

- *2 TaskCPU 40
- @20: *2 TaskCPU 40
- @30: *2 TaskCPU 30

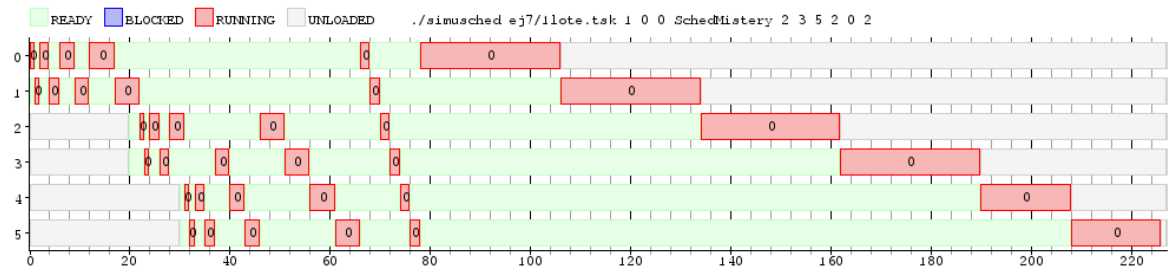


Figure 11: Scheduler Mistery

El costo de cambio de contexto y costa de migración de núcleo se decidió mantenerlo en 0, ya que no afecta el comportamiento del scheduler. En el gráfico (nombre del grafico) se ve que las tareas se encolaron en la primera *colaReady*, ejecutan durante un solo tick y luego se encolan en la cola siguiente, que es la que tiene un quantum asociado igual a 2. El comportamiento sigue hasta que en el instante 20 llegan dos tareas nuevas. Cuando la tarea 1 termina su quantum (instante 23), se empiezan a ejecutar las tareas en las colas con mayor prioridad, es decir las que acaban de llegar. Lo mismo sucede cuando llegan las tareas 4 y 5 en el el ciclo 30 de la simulación. Notar que las tareas 0 y 1 no vuelven a obtener

uso del cpu hasta el tick número 66. Esto es porque el scheduler le da prioridad a las tareas mas nuevas, ya que se encolan en las colas con mayor prioridad. Esta característica podría hacer que las tareas mas "viejas" tarden demasiado tiempo en obtener el cpu si continuamente estan llegando nuaves tareas al sistema. Otra característica a observar es que efectivamente cuando se le asigna una tarea a la cola con quantum asociado 0, esta va a correr hasta hacer EXIT.

Para poder estudiar con mayor facilidad el comportamiento del scheduler cuando una tarea se bloquea, se creó dos tareas $Task1$ y $Task2$, que no reciben ningún parámetro, hacen uso de cpu y realizan llamadas bloqueantes.

El lote de tareas es:

- Task2BLOCK
- Task1BLOCK
- @15: TaskCPU 20

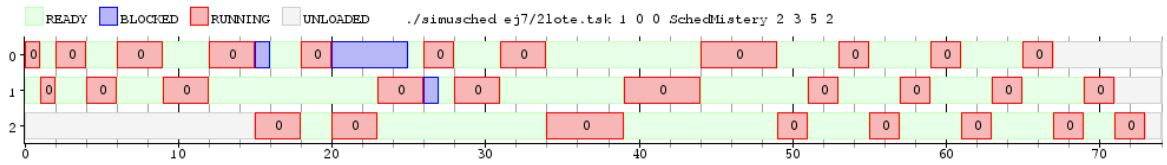


Figure 12: Scheduler Mistery

En el instante 0 llegan dos tareas. La tarea 0 es $Task2BLOCK$ y la tarea 2 es $Task1BLOCK$. En el ciclo de clock número 15 llega la $TaskCPU$ y las dos primeras tareas ya ejecutaron el quantum correspondiente a las tres primeras colas. En el instante 12, la tarea 0 se encuentra en la cola de con cantidad de quantum asociado 5. Pero en el ciclo 14 realiza una llamada bloqueante por lo que el scheduler la desaloja, y pasa a ejecutar a la tarea 2 que fue la última en llegar al sistema. Notar que como no hay ninguna otra tarea en las colas con quantum 1 y 2, solo la tarea 2 hace uso exclusivo del cpu del momento 15 al 18. Pero como al bloquearse una tarea, el scheduler la encola en la cola anterior a la que estaba, ahora pasa a ejecutar la tarea 0 nuevamente. De nuevo realiza otra llamada bloqueante, por lo que ahora esta tarea va a pasar a la cola con quantum 2. Esto lo podemos observar en el instante 26, la tarea 0 se ejecuta dos ciclos de clock. despues una vez que vuelve a hacer uso del cpu, realiza 3 ciclos y por ultimo pasa a la cola de quantum 5 y luego quantum 2 hasta que finaliza. La tarea 1 también se realiza una llamada bloqueante en el tick número 25 en el momento que estaba en la cola de quantum 5. Se puede ver que la siguiente vez que ejecuta, se le asigna 3 ciclos de clock, que era lo esperado.

8 Ejercicio 8

En el ejercicio 8 se pide realizar una implementación del scheduler round robin, pero sin migración de procesos entre núcleos. Para esto se agregaron nuevas estructuras a la clase SchedRR2:

- **colaReady**: Ahora es un vector de colas, las cuales van a almacenar los pid de las tareas que ultilicen el core correspondiente.
- **cantTasks**: Indica la cantidad de tareas que tiene cada cpu (RUNNING + BLOCKED + READY).
- **cpuTareasBloqueadas**: Es una cola de tuplas con un pid en la primera coordenada (proceso que hizo llamada bloqueante) y número de core en el cual estaba ejecutando.

Los métodos load, unblock y tick también cambian respecto del round robin con migración de procesos.

- **load(pid)**: Se encarga primero de buscar cual es el cpu con menor cantidad de tareas y, una vez encontrado, sumar uno a la cantTasks de ese cpu. Luego encola la tarea en la cola correspondiente de tareas ready.
- **unblock(pid)**: Como se pide que no haya migración de procesos entre núcleos, se guarda en la estructura *cpuTareasBloqueadas* todos las tareas bloqueadas con su número de core. Esta función encuentra la tupla en la cola, la quita de la cola, y luego encola la tarea en la colaReady del cpu obtenido.
- tick(cpu, m): Motivo TICK sigue siendo similar a la implementación anterior. Motivo BLOCK ademas de quitar al pid de la *colaReady* del cpu, arma una tupla de la forma (pid, cpu), siendo pid = current_pid(cpu). Luego encola esa tupla en *cpuTareasBloqueadas*, si quedan tareas en la *colaReady* toma la primera y devuelve su pid, y sino hay mas ready, devuelve IDLE_TASK. Motivo EXIT acá se resta uno a la cantTasks del cpu actual, y se devuelve la tarea que sigue en la *colaReady* y si no hay mas, IDLE_TASK.

Para corroborar su correcto funcionamiento se ideó un lote de tareas que permita chequear que, efectivamente no haya migración de núcleos y que se asignen correctamente los cores que cada tarea va a utilizar.

Lote utilizado:

```
TaskCPU 20
TaskConsola 20 1 5
TaskCPU 5
@2:
*2 TaskCPU 20
TaskCPU 5
@4:
TaskCPU 20
TaskConsola 10 1 5
TaskConsola 2 1 2
@18:
TaskCPU 5
TaskConsola 3 2 4
```

La simulación se hizo con 3 cores con cambio de contexto 0, ya que no modifica el comportamiento del scheduler, y el costo de migración de núcleo 0. La idea del lote, es cargar los primeros cores con tareas que tarden más en terminar, y dejar en el último tareas que vayan terminando y que liberen el cpu, para poder apreciar de esta forma, que efectivamente se asignará el cpu con menos tareas. Debido a la implementación, cuando todos los cores tengan igual cantidad de procesos, se le asignará el cpu más chico numéricamente.

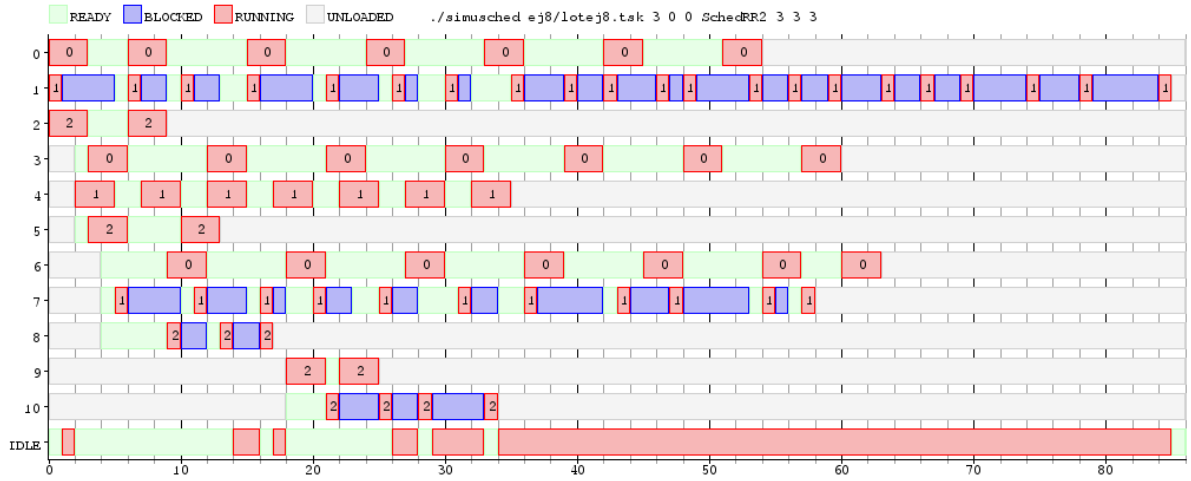


Figure 13:

Se puede observar que no existe migración de núcleos, ya que todas las tareas mantienen su número de cpu a lo largo de toda la simulación. En los instantes 0, 2 y 4 llegan las tareas y se les asignan los cores de la forma esperada. En el instante 4, los 3 cpu tienen 3 tareas cada uno. En el instante 15 el cpu 2 se queda sin tareas para ejecutar(la tarea 2 termina en el momento 2, la 5 en el 9 y la 13 en el 15). Por lo tanto se le asigna el IDLE_TASK por un ciclo de clock hasta que llegan dos nuevas tareas. Vemos que efectivamente se le asignan al core número 2, que es el que menos tareas tenía de los tres. En el tick 32, la tarea 10 hace EXIT y el cpu 2 no tiene mas tareas que ejecutar. Este representa una gran desventaja, ya que mientras los dos primeros cpu están trabajando y tienen tareas esperando, el cpu 2 permanece inactivo. Esto pasa porque el único método de equilibrado de carga que se implementa es comprobar que cpu tiene menos tareas. Para mejorar el uso de cpu o equilibrar la cantidad de trabajo de cada core, se podría implementar la *migración solicitada*, que hace que se pasen tareas a un cpu que esté inactivo (sin tareas bloqueadas).

Por otro lado, matener un proceso siempre en un mismo núcleo tiene sus ventajas. Por ejemplo, si se considera lo que ocurre con la memoria caché cuando una tarea se estuvo ejecutando en un cpu específico: los datos a los que el proceso ha accedido mas recientemente se almacenan en la caché de ese procesador, luego gran parte de accesos a memoria se evitan ya que los datos se encuentran en la caché. Además si se produjera una migración de núcleo, los datos de esa caché deberían invalidarse, y luego debería llenarse la caché del nuevo cpu. Este concepto se lo llama $afinidad\ al\ procesador$, que es que cada proceso tiene una afinidad al cpu en el que esta ejecutando. En esta implemetación, se utiliza $afinidad\ dura$, que no permite la migración entre cores. También existe la $afinidad\ suave$, en este caso es posible que una tarea cambie de cpu.

Para ver el beneficio de la $afinidad\ dura$ se construyó el siguiente lote de tareas:

*4 TaskCPU 100

Se realizó una simulación con la primera implementación de Round Robin, que permitía la migración, y una con la que no. Para tener en cuenta los costos de tiempo de invalidar la caché y rellenarla cada vez que se cambie de núcleo, se estableció 4 ticks como costo de migración de cpu. Hay que tener en cuenta que el simulador no provee funciones que permitan emular el uso de la memoria caché, es decir, no se va a poder medir cantidad de accesos a memoria y accesos a chaché. La simulación se realizó con 3 cores, y todos con misma cantidad de quantum.

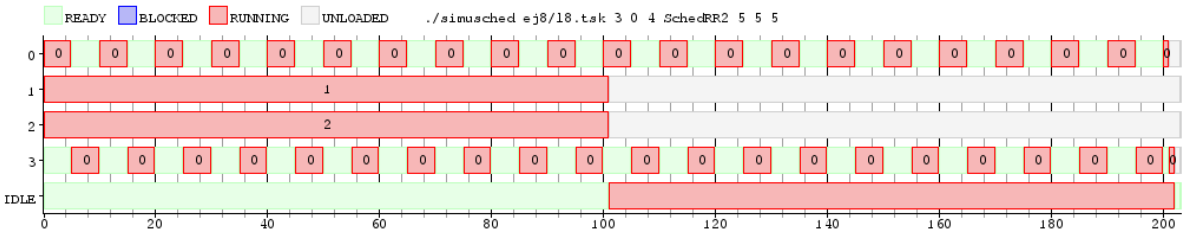


Figure 14: Round Robin Sin migración.

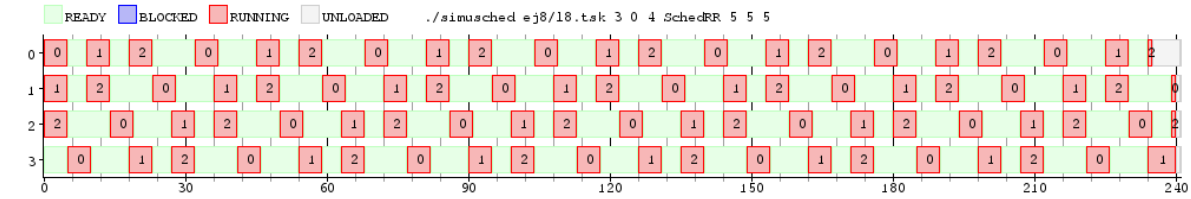


Figure 15: Round Robin Con migración.

A simple vista, se ve como el *tiempo de ejecución* mejoró respecto de la primera implementación de Round Robin. En el gráfico 15 se ve que recién en el instante 234 termina la primera tarea del lote y en el 240 las últimas 2. En el gráfico 14 vemos que las dos útlimas tareas terminan en el tick 205 aproximadamente. Notar también como a partir del instante 100 del gráfico 14, tanto el cpu 1 como el 2 quedan inactivos por mas de 100 ciclos. Esta desventaja se describió en el primer ejemplo.