

Trabajo Práctico de Sistemas Operativos

30 de octubre de 2014

Universidad de Buenos Aires - Departamento de Computación - FCEN

Integrantes:

- Castro, Damián L.U.: 326/11 ltdicai@gmail.com
- Toffoletti, Luis L.U.: 827/11 luis.toffoletti@gmail.com
- Zanollo, Florencia L.U.: 934/11 florenciazanollo@gmail.com

Índice

1. Introducción	3
1.1. Simulador	3
1.1.1. Acciones del simulador	4
1.1.2. Definición de <i>schedulers</i>	4
2. Tareas	4
2.1. TaskCPU	4
2.2. TaskIO	4
2.3. TaskAlterno	5
2.4. TaskConsola	5
2.5. TaskBatch	6
3. Políticas de <i>scheduling</i>	7
3.1. First-Come First-Served (FCFS)	7
3.2. Round-Robin (RR)	8
3.2.1. Código	8
3.2.2. Simulaciones y análisis	8
3.3. Round-Robin sin migración (RR2)	10
3.3.1. Contra. Caso: Sobrecarga de un núcleo	10
3.3.2. Pro. Caso: Migración costosa	11
3.4. Lottery Scheduling	12
3.4.1. Loterías	12
3.4.2. Fairness	12
3.4.3. Compensation tickets	13
4. Discusión	15
4.1. Métricas	15
4.1.1. Fairness	15
4.1.2. Response time	17

1. Introducción

Una computadora moderna consiste de uno o más procesadores, memoria, discos, impresoras y uno o varios dispositivos de entrada/salida. Además una computadora ejecuta programas que suelen intentar acceder a estos recursos y generalmente asumiendo que son los únicos que desean utilizarlos. Sin embargo esto casi nunca es cierto y suelen haber conflictos cuando dos procesos (programas en ejecución) acceden al mismo recurso. Para solucionar estas problemáticas se recurre a los sistemas operativos, que son aquellos que se encargan de coordinar el acceso a los recursos por parte de los procesos. Un sistema operativo es un proceso superior a todos los demás y decide, siguiendo alguna normativa, en que momentos los procesos comunes se ejecutan. Se conoce a esta normativa como política de *scheduling*, y no hay una única manera de definirla. Dependiendo de la situación puede que se prefiera políticas que permitan a los procesos mantener ocupado un recurso durante períodos largos de tiempo mientras que otras políticas se valen en desalojar a las tareas y otorgarle los recursos a otra tarea. Se espera que un *scheduler* trate de cumplir los siguientes objetivos:

- Ecuanimidad (*fairness*): que cada proceso reciba una dosis “justa” de CPU (para alguna definición de justicia).
- Eficiencia: tratar de que la CPU esté ocupada todo el tiempo.
- Carga del sistema: minimizar la cantidad de procesos listos que están esperando CPU.
- Tiempo de respuesta: minimizar el tiempo de respuesta percibido por los usuarios interactivos.
- Latencia: minimizar el tiempo requerido para que un proceso empiece a dar resultados.
- Tiempo de ejecución: minimizar el tiempo total que le toma a un proceso ejecutar completamente.
- Rendimiento (throughput): maximizar el número de procesos terminados por unidad de tiempo.
- Liberación de recursos: hacer que terminen cuanto antes los procesos que tiene reservados más recursos.

Como se puede intuir, es imposible cumplir todos los objetivos a la vez, por lo tanto las políticas de *scheduling* tienen sus ventajas y sus desventajas. Es el enfoque de este informe exhibir algunas de las políticas ms conocidas, su implementación en lenguaje *C++* y mostraremos sus pros y sus contras con ejemplos y incluiremos gráficos para ayudar a la comprensión.

1.1. Simulador

Para analizar las diferentes políticas de *scheduling* utilizamos un modelo de computadora simplificado que consiste en los siguientes componentes:

- Un lote de programas a ejecutar.
- Uno o varios núcleos (*cores*) de procesamiento, que se encargarán de hacer los cálculos de los procesos.
- Un único dispositivo de entrada y salida que todos los procesos pueden utilizar si lo desean.

Además contamos con un simulador de ese modelo, que nos permite ejecutar programas o tareas de acuerdo a un *scheduler* dado. El mismo se encarga de imprimir una secuencia de eventos resultantes de la ejecución.

1.1.1. Acciones del simulador

El simulador provee a los procesos con tres acciones posibles:

- *uso_CPU*(*n*) que indica que el proceso utilizará el CPU durante *n* ciclos de reloj.
- *uso_IO*(*n*) que indica que el proceso utilizará un recurso de entrada/salida durante *n* ciclos de reloj. Durante este tiempo el proceso no utiliza al procesador, así que el *scheduler* es libre de utilizar ese tiempo en otra tarea.
- *return* que indica que el proceso ya terminó su ejecución. El *scheduler* debe realizar las acciones necesarias para desalojar al proceso.

1.1.2. Definición de *schedulers*

Un *scheduler* válido ofrece las siguientes funcionalidades:

- *load(pid)*: Carga un proceso con número de identificación *pid*.
- *unblock(pid)*: Se realiza cuando un proceso con número de identificación *pid* deja de utilizar un recurso de entrada/salida y desea volver a usar el procesador.
- *tick(cpu, motivo)*: Se ejecuta por cada *tick* del reloj del procesador *cpu* y el *motivo* indica que sucedió con el último proceso que estaba en ejecución. Un *motivo* puede valer tres cosas:
 - TICK: indica que el proceso consumió todo el ciclo.
 - BLOCK: indica que el proceso pidió acceso a un recurso de entrada y salida y por este motivo la tarea fue bloqueada.
 - EXIT: indica que la tarea terminó de ejecutarse.

2. Tareas

2.1. TaskCPU

Simula la utilización del CPU.

Algorithm 1 TaskCPU

```
1: procedure TASKCPU(n)  
2:   uso_CPU(n) ▷ Utiliza al CPU durante n ciclos.  
3: end procedure
```

2.2. TaskIO

Simula la utilización del CPU durante un tiempo y realiza una llamada bloqueante para acceder a un recurso de entrada/salida.

Algorithm 2 TaskIO

```
1: procedure TASKIO(n, m)  
2:   uso_CPU(n) ▷ Utiliza al CPU durante n ciclos.  
3:   uso_IO(m) ▷ Realiza una llamada bloqueante que durará m ciclos.  
4: end procedure
```

2.3. TaskAlterno

Alterna entre usar el CPU y llamadas bloqueantes.

Algorithm 3 TaskAlterno

```
1: procedure TASKALTERNO( $n_1, n_2, \dots, n_k$ )
2:   for  $i \leftarrow [1..k]$  do
3:     if esPar( $i$ ) then                                ▷ Alterna entre utilizar el CPU o realizar llamadas bloqueantes.
4:        $uso\_CPU(n_i)$ 
5:     else
6:        $uso\_IO(n_i)$ 
7:     end if
8:   end for
9: end procedure
```

2.4. TaskConsola

Algorithm 4 TaskConsola

```
1: procedure TASKCONSOLA( $cant\_bloqueos, bmin, bmax$ )
2:   seteo semilla de random
3:   for  $i \leftarrow 0, cant\_bloqueos$  do
4:      $random\_number \leftarrow modulo(rand(), bmax - bmin + 1) + bmin$                                 ▷ [1]
5:      $Uso\_IO(random\_number)$ 
6:   end for
7: end procedure
```

[Nota 1]: `rand()` retorna un número pseudoaleatorio arbitrariamente entre 0 y `RAND_MAX`, una constante del lenguaje. Vale entonces que si:

$$\begin{aligned} rand() \in [0..RAND_MAX] &\implies rand() \bmod (bmax - bmin + 1) \in [0..bmax - bmin] \\ &\implies rand() \bmod (bmax - bmin + 1) + bmin \in [bmin..bmax] \end{aligned}$$

2.5. TaskBatch

Realiza una cantidad determinada de llamadas bloqueantes en momentos pseudoaleatorios.

Algorithm 5 TaskBatch

```
1: procedure TASKBATCH(total_cpu, cant_bloqueos)
2:   vector<bool> bloqueos  $\leftarrow$  elegirMomentosParaBloquear(total_cpu, cant_bloqueos)  $\triangleright$  [Nota 2]
3:   for  $i \leftarrow [1..total\_cpu]$  do  $\triangleright$  n veces
4:     if bloqueos[ $i$ ] == true then
5:       uso_IO(1)
6:     else
7:       uso_CPU(1)
8:     end if
9:   end for
10: end procedure
```

[Nota 2]: *elegirMomentosParaBloquear*(n, m) elige m momentos entre $[1..n]$ de manera pseudoaleatoria y retornar un vector de *booleanos* que indicarán cuándo realizar una llamada bloqueante.

3. Políticas de *scheduling*

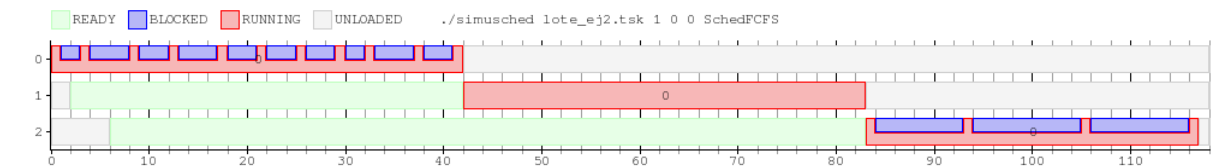
3.1. First-Come First-Served (FCFS)

FCFS (First-Come First-Served) es un scheduler simple en el cual los procesos son asignados al CPU en el orden en que estos lo requieren.

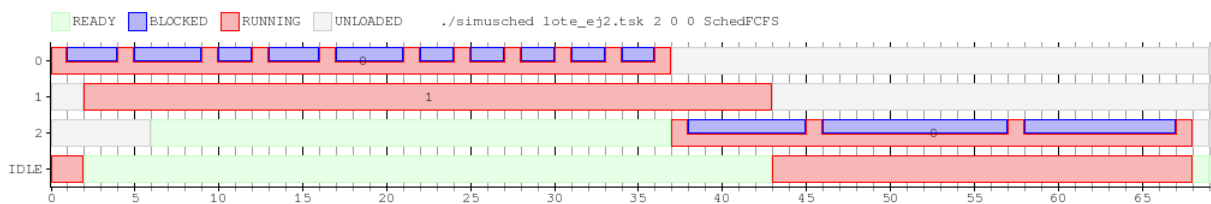
```
TaskConsola 10 2 4
@2:
TaskCPU 40
@6:
TaskConsola 3 6 11
```

Básicamente hay una sola cola (FIFO) de procesos 'Ready'. Cuando un proceso requiere CPU y éste está libre, se lo deja correr tanto tiempo como quiera y sin interrupciones.

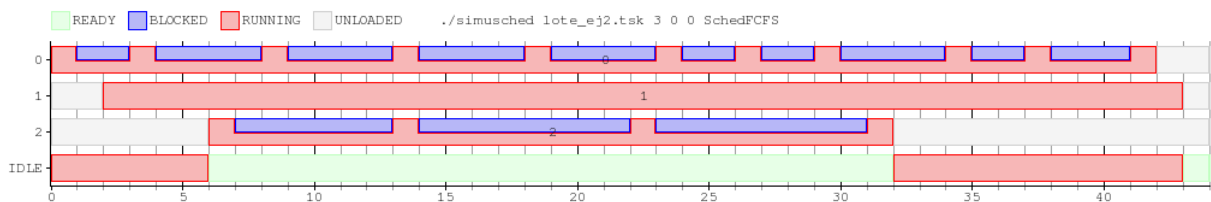
A continuación se muestran los gráficos para 1, 2 y 3 núcleos, usando SchedFCFS para el lote de tareas del cuadro.



2



3



Obs: Los parámetros de costo de cambio de contexto y migración fueron seteados en 0 ya que son irrelevantes para este scheduler. Porque nunca se desalojan las tareas ni se cambian de núcleo. Sólo serían usados por el simulador al momento de cargar un proceso por primera vez

En los gráficos se hace evidente que a más cantidad de núcleos, mejor rendimiento. Esto se debe a que FCFS no soporta multitarea por sí sólo (ya que nunca desaloja a las tareas), sino que necesita varios núcleos para lograrlo.

Las desventajas de FCFS son varias:

- Como ya dijimos, no soporta multitarea.
- Puede generar mucho 'waiting time'. Si está ejecutando tareas muy largas las nuevas no entran hasta que éstas terminen.
- No tiene buen rendimiento, a menos que se sepa la duración de las tareas de antemano.
- Carece de 'fairness' i.e. no distribuye el/los procesador/es de forma justa entre las tareas.

3.2. Round-Robin (RR)

RR es un algoritmo de los más viejos, simple, justo y ampliamente usado.

A cada proceso se le asigna un tiempo (quantum) durante el cuál le es permitido correr. Si el proceso sigue corriendo al final del quantum, los recursos le son quitados y se le da paso a otro proceso. Si el proceso se bloquea, pierde el quantum que le quedaba y los recursos son dados a otro proceso.

Implementar RR es simple, sólo se requiere una lista de procesos 'Ready'. Cuando el proceso usa su quantum es puesto al final de la lista. Si éste se bloquea, es removido de dicha lista y volverá a ser agregado en cuanto se desbloquee.

3.2.1. Código

En nuestro caso usamos una cola de procesos llamada 'readyTasks'.

Algorithm 6 Round-Robin

```
1: procedure LOAD(pid)
2:   encolo la nueva tarea en readyTasks
3: end procedure
4: procedure UNBLOCK(pid)
5:   encolo la tarea desbloqueada en readyTasks
6: end procedure
7: procedure TICK(cpu, motivo)
8:   if el cpu esta ejecutando IDLE then
9:     llamo a NEXT(cpu) para obtener la próxima tarea a ejecutar
10:  else
11:    if el motivo es un TICK then
12:      Actualizo el quantum
13:      if se termino el quantum then
14:        encolo la tarea en readyTasks
15:        renuevo el quantum
16:        llamo a NEXT(cpu) para obtener la próxima tarea a ejecutar
17:      end if
18:    else if el motivo es un BLOCK o un EXIT then
19:      renuevo el quantum
20:      llamo a NEXT(cpu) para obtener la próxima tarea a ejecutar
21:    end if
22:  end if
23: end procedure
24: procedure NEXT(cpu)
25:   if hay tareas esperando en readyTasks then
26:     retornar la primera de la cola readyTasks y quitarla
27:   else
28:     retornar IDLE
29:   end if
30: end procedure
```

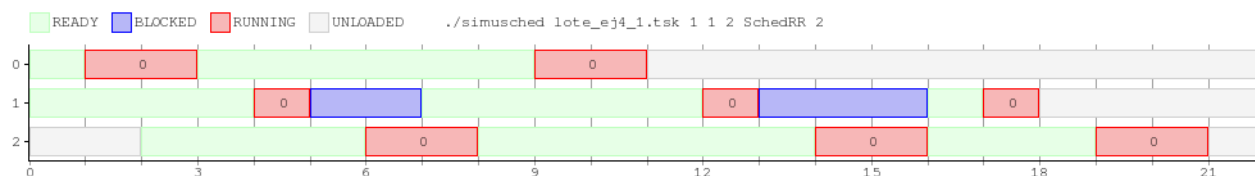
3.2.2. Simulaciones y análisis

Para las simulaciones mantuvimos los siguientes costos:

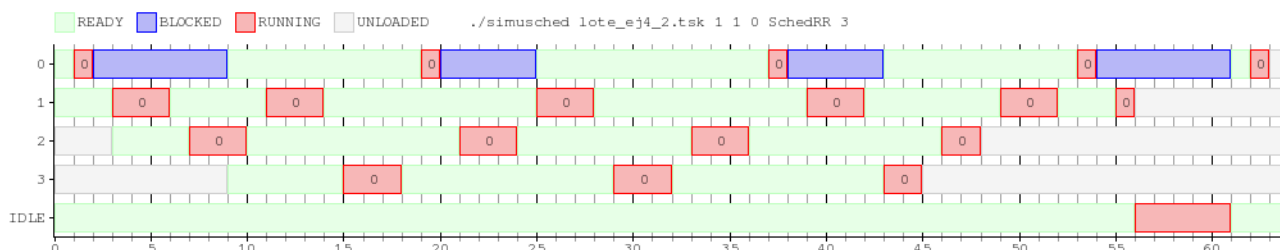
costo de cambio de contexto = 1 : En un CPU real se deben intercambiar estructuras de datos que contienen información de los procesos antes de poder correr la nueva tarea.

costo de migración de núcleo = 2 : Se deben intercambiar las mismas estructuras de datos que para el cambio de contexto pero la 'distancia' del caché de un núcleo al de otro es mayor que dentro de sí mismo.

Correremos un par de lotes de tareas para verificar que el comportamiento del scheduler es el esperado.



Se puede observar que no es un algoritmo FCFS porque, en principio, hace desalojo de tareas cada cierto quantum. Además las tareas que se bloquean son desalojadas de inmediato, sin encolarlas en ReadyTasks hasta que se desbloqueen.



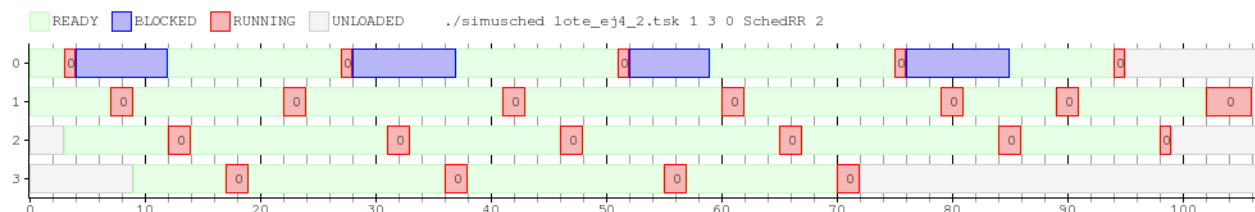
Se puede ver que el proceso 0 es desalojado en cuanto se bloquea, cediendo el procesador al proceso 1 (siguiente en la cola de ReadyTasks).

En cuanto el proceso 1 termina su quantum es desalojado (y movido al final de la cola de ReadyTasks). Y se deja correr el proceso 2.

Entre los ciclos 14 y 15, la cola de ReadyTasks tiene la siguiente forma: [3, 2, 0, 1].

Ya que el proceso 0 fue desencolado en cuanto se bloqueó, y vuelto a encolar en el ciclo 11 (cuando se desbloquea), antes de que termine el quantum del proceso 1.

Veamos ahora un ejemplo de una mala elección de quantum para un entorno donde el costo de cambio de contexto es elevado.



Como se puede observar, tarda 106 ciclos en completar la ejecución del lote de tareas. A diferencia del anterior (que tardaba 63 ciclos).

Esto se debe a que el procesador está más tiempo haciendo tareas de mantenimiento (cambio de contexto) que efectivamente ejecutando procesos.

3.3. Round-Robin sin migración (RR2)

RR sin migración es una variante del RR que no permite la migración de procesos entre los núcleos.

En este algoritmo cuando un proceso nuevo empieza a correr le es asignado un núcleo, el que tenga menos procesos asignados (contando procesos en espera, corriendo y bloqueados). Luego, durante toda su ejecución el proceso está ligado a ese núcleo.

Esta variante, como todo algoritmo de scheduling, tiene sus pros y sus contras. Para simplificar las explicaciones y los gráficos tomamos ejemplos de dos núcleos, aunque esto puede ser expandido a más de dos.

3.3.1. Contra. Caso: Sobrecarga de un núcleo

Una contra es que uno de los núcleos puede quedar sobrecargado mientras el otro no tiene ningún proceso asignado (está corriendo IDLE).

El lote de tareas utilizado tiene dos posibles procesos, uno que utiliza CPU durante muchos ciclos (proceso largo) y otro que utiliza CPU durante unos pocos ciclos (proceso corto). Son en total 6 procesos, por simplicidad.

Si los procesos llegan alternándose entre largos y cortos entonces:

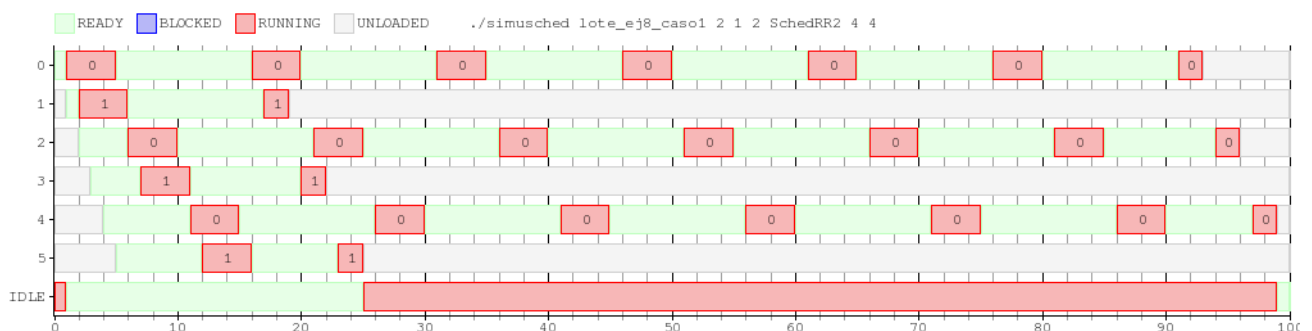
- Al primer proceso (largo) le es asignado el núcleo 0.
 - Al segundo proceso (corto) se le asigna el núcleo 1.
 - Al tercer proceso (largo) le es asignado el núcleo 0.
- ... y continúa así.

Con lo cual, luego de la carga del último proceso quedan distribuidos de la siguiente manera:

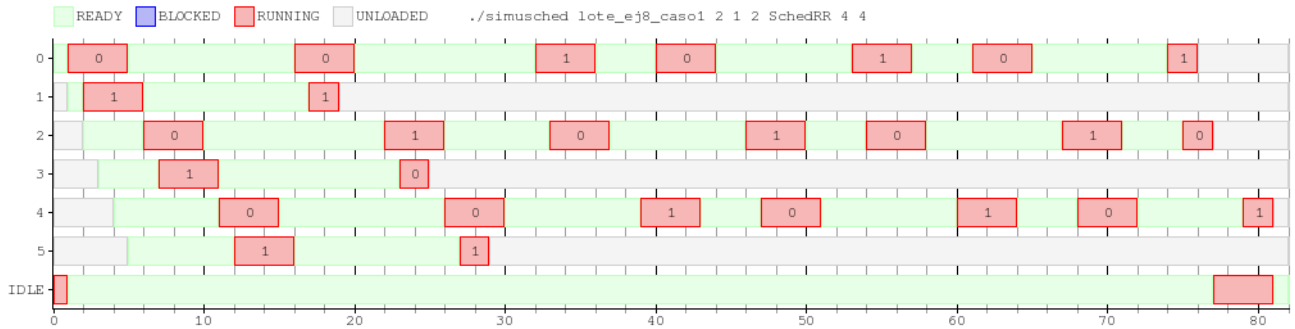
Núcleo 0: 3 procesos largos. Núcleo 1: 3 procesos cortos.

En el ciclo 25 los procesos cortos son finalizados quedando el núcleo 0 con tres procesos y el 1 IDLE.

RR2 (sin migración)



RR (con migración)



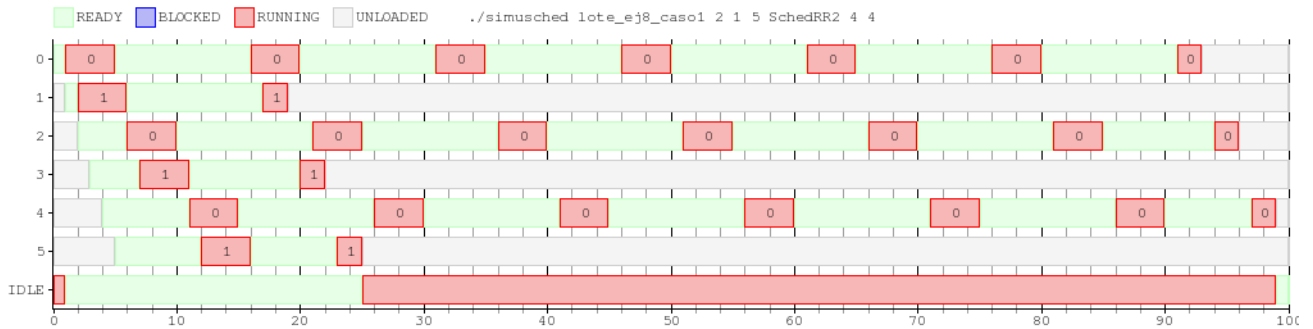
Este escenario también se podría dar si en vez de tareas cortas tenemos tareas que realizan llamadas bloqueantes. Entonces en el núcleo 0 hay tareas usando CPU y el núcleo 1 queda esperando que algún proceso se desbloquee.

3.3.2. Pro. Caso: Migración costosa

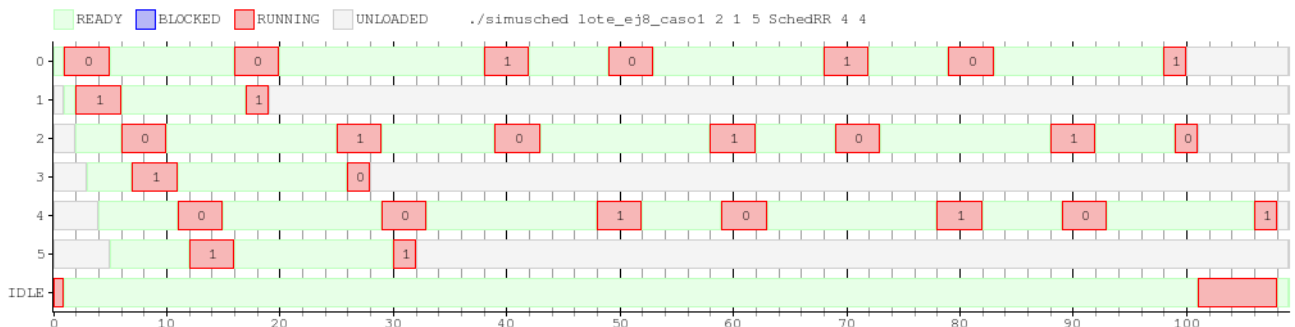
Una de las ventajas de RR2 queda expuesta si el costo de la migración es grande. Si los procesos cambian mucho entre núcleos puede que el procesador pase más tiempo haciendo tareas de mantenimiento que efectivamente corriendo procesos. Tal es el caso de RR.

Vamos a correr el experimento sobre el mismo lote de tareas y con los mismos quantums que en el caso anterior, para que sea más obvio el cambio. Lo único que vamos a variar es el costo de migración que va a pasar de 2 a 5.

RR2 (sin migración)



RR (con migración)



No sólo RR2 termina apróx 10 ciclos antes; incluso bajo este lote de tareas que es un caso negativo; sino que además RR pasa mucho tiempo haciendo tareas de mantenimiento.

3.4. Lottery Scheduling

La idea detrás de *lottery scheduling*[2] es aplicar un enfoque probabilístico y pseudoaleatorio a la hora de asignar recursos a las tareas. Para lograr esto se utiliza un sistema de *tickets* que representan derechos de acceso a recursos; las tareas poseen una cantidad determinada de *tickets*, pudiendo ser distinta de una tarea a otra y cada vez que el *scheduler* decide a qué tarea otorgar los recursos lo hace mediante una lotería, elige uno entre todos los *tickets* del sistema. El proceso que tenga ese *ticket* será el ganador y se le asignará el recurso que está demandado.

Los *tickets* son una forma de cuantificar los derechos de acceso a recursos; cuantos más *tickets* tenga una tarea, más probable es que gane la lotería. También permiten homogeneizar los diferentes tipos de recursos ya que todos son representados por los mismos tipos de *tickets*, esto aporta flexibilidad a la hora de asignarlos.

3.4.1. Loterías

En cada *tick* del reloj, el *scheduler* toma una decisión: si el proceso en ejecución agotó todo su quantum asignado, realizó una llamada bloqueante o simplemente terminó exitosamente entonces se lleva a cabo una lotería. Esta lotería decide quién va a ser el próximo proceso en utilizar el recurso deseado. Elegir el *ticket* ganador se realiza mediante un algoritmo pseudoaleatorio que lo elige de manera uniforme, esto es, todos los tickets tienen una probabilidad igual de ganar la lotería, a saber:

Sea n la cantidad de *tickets* y $\{t_i\}_{1 \leq i \leq n}$ el conjunto de ellos.

$$P(t_i) = \frac{1}{n}$$

Luego, la probabilidad que un proceso j que posee t cantidad de *tickets* gane la lotería es igual a:

$$P(j \text{ gane la lotería}) = \sum_1^t \frac{1}{n} = \frac{t}{n}$$

Se puede probar que la cantidad de loterías ganadas por la tarea j sigue una distribución binomial de parámetro $p = \frac{t}{n}$. De esta manera se observa que la esperanza de la cantidad de loterías ganadas por una tarea es igual a:

$$E(\text{cantidad de loterías ganadas}) = \frac{t}{n} * \text{cantidad de loterías realizadas}$$

Esto implica que el valor promedio de la cantidad de loterías ganadas es directamente proporcional a la cantidad de *tickets* que posee la tarea. Por esta razón se dice que *lottery scheduling* es probabilísticamente justo.

3.4.2. Fairness

Para medir la ecuanimidad del algoritmo del scheduler, diseñamos un experimento que analiza los ciclos consumidos por cada tarea por intervalos a lo largo del tiempo. El SchedLottery que implementamos tiene como objetivo otorgar el mismo tiempo de uso de CPU para todas las tareas, es decir que si pongo a correr dos tareas ambas se ejecuten una cantidad de tiempo muy similar (relación 1:1).

Entonces si nuestra implementación cumple la ecuanimidad, deberíamos observar que a medida que avanza la ejecución, los tiempos de CPU de cada tarea coinciden en un mismo valor o uno muy cercano.

Realizamos la prueba con una TaskBatch y una TaskCPU, realizando 1000 ejecuciones:

SchedLottery:

Ciclo	10	20	30	40	50	60	70	80	90	100	110	120	130	140
Tarea 0 (CPU)	3	8	11	15	18	22	25	28	31	35	39	44	49	51
Tarea 1 (Batch)	2	5	7	11	14	17	21	24	27	30	34	40	47	51

Se puede observar que los valores mantienen el mismo grado de magnitud a lo largo de la ejecución. A continuación evaluamos las mismas tareas bajo un scheduler Round-Robin, que por definición es un algoritmo que posee mucha ecuanimidad, con el que podremos comparar nuestros resultados previos.

Round-robin:

Ciclo	10	20	30	40	50	60	70	80	90	100	110	120	130	140	146
Tarea 0 (CPU)	4	9	12	16	21	24	28	33	38	42	45	50	51	-	-
Tarea 1 (Batch)	3	6	10	13	16	20	23	25	27	30	34	37	41	47	51

Como se puede apreciar la brecha de utilización de CPU para RR crece en favor de la TaskCPU, esto se debe a que no posee un sistema para compensar cuando tenemos tareas que se bloquean y otras que no, como si posee el SchedLottery (ver sección Compensation Tickets).

Dado los resultados de los experimentos pudimos mostrar que el SchedLottery no se inclina en favor de ninguna de las tareas, distribuyendo el uso del CPU uniformemente cumpliendo con la definición de fairness.

3.4.3. Compensation tickets

Queremos ver cómo se comporta el algoritmo ante un lote de tareas compuesto por tareas que sólo utilizan CPU y otras que realizan llamadas bloqueantes.

Si SchedLottery no contara con compensation tickets entonces como se explica en el artículo [2, Sec. 3.4] el problema se ocasiona cuando una tarea se bloquea sin terminar de utilizar su quantum. De continuar así por varios ciclos entonces esa tarea usaría, en promedio, menos CPU que las demás (la probabilidad de ganar el sorteo es la misma para todas, pero en este caso habría tareas que usan más tiempo el CPU que las demás) violando así la regla de 1:1 dónde todas deberían recibir la misma proporción de CPU.

La forma propuesta para contrarrestar este problema es Compensation Tickets. Consiste en dar a la tarea bloqueada una cantidad de tickets en relación con el quantum no utilizado. Así, si sólo utilizó una fracción F del quantum le son dados $\frac{CANT_INICIAL}{F}$ tickets.

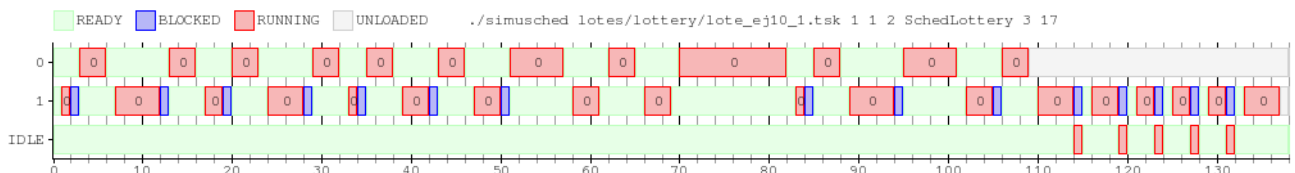
Queremos mostrar evidencia de que el sistema de compensation tickets (CT) corrige el problema previamente mencionado sobre la asignación de CPU esperada entre tareas bloqueantes y no bloqueantes. Para esto ejecutamos dos tareas, una taskBatch, ya que el problema se manifiesta cuando existen llamadas bloqueantes, y la otra taskCPU.

Al comienzo cada tarea posee 10 tickets (misma probabilidad de ganar) porque queremos que se ejecuten equitativamente.

Para mostrar que efectivamente hay una mejora en la distribución del CPU presentamos los datos mediante tablas, donde cada columna pertenece a un ciclo en particular y cada fila a una tarea.

Ej: La posición de la tabla fila x, columna y: tiene la cantidad de CPU utilizada por la tarea x hasta el ciclo y.

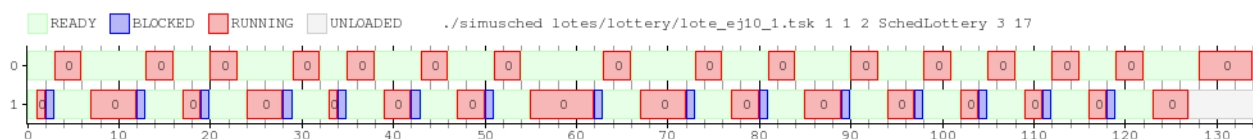
SchedLottery SIN CT:



Ciclo	10	20	30	40	50	60	70	80	90	100	110	120	130	137
Tarea 0 (CPU)	3	6	10	15	18	24	27	37	42	47	51	-	-	-
Tarea 1 (Batch)	4	8	12	14	19	21	25	25	27	31	34	41	46	51

Hasta el ciclo 80 el CPU se mantiene relativamente distribuido. Luego la tarea 0 pasa a tener mayor proporción hasta que finaliza. Si esto se proyecta en el tiempo entonces la brecha se vuelve cada vez más grande. Rompiendo así con el 1:1 deseado.

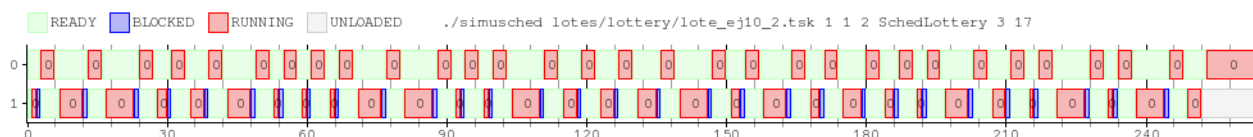
SchedLottery CON CT:



Ciclo	10	20	30	40	50	60	70	80	90	100	110	120	130	134
Tarea 0 (CPU)	3	6	10	15	18	21	24	27	30	35	39	43	47	51
Tarea 1 (Batch)	4	8	12	14	19	24	29	34	38	41	44	47	51	-

Ahora con CT vemos que la tarea 0 nunca toma la delantera en cuanto al CPU. Esto se debe a que la tarea 1 tiene más probabilidades de ganar, cada vez que no utiliza todo su quantum esta probabilidad se hace más grande. Así si están las dos tareas esperando, es muy probable que la 1 sea la elegida para correr a continuación. Si esto se proyecta en el tiempo se continúa con la proporción 1:1.

Esto es un ejemplo de proyección en el tiempo:



Ciclo	30	60	90	120	150	180	210	240	252	264
Tarea 0 (CPU)	9	21	32	43	54	63	75	87	90	101
Tarea 1 (Batch)	14	24	36	47	60	73	84	94	101	-

4. Discusión

4.1. Métricas

Para observar el comportamiento de los algoritmos de *scheduling* no basta con hacer un gráfico, también hay que poder extraer información de él y así realizar un análisis del rendimiento del algoritmo. Esa información tiene que poder ser cuantificable, ya que si lo fuese podríamos utilizar esta información para comparar varios algoritmos y decidir cuál es el más conveniente según un conjunto determinado de procesos.

Por ello recurrimos al uso de métricas, para poder obtener números a partir de una corrida determinada de un algoritmo de *scheduling*. Las métricas nos permitirán decidir si el algoritmo está aportando a cumplir algunos de los objetivos indicados anteriormente en la introducción. Tanenbaum[1] propone una división entre el tipo de tareas posibles, en base a sus características, a saber:

Tipos de tareas:

- *Batch* o de lotes, que se caracterizan por utilizar mucho el CPU, de manera continua y con pocos accesos a dispositivos de entrada y salida.
- *Interactive* o interactivos, que particularmente usan mucho los dispositivos de entrada y salida.

Los conjuntos de tareas que utilizamos para probar los algoritmos de *scheduling* en el simulador generalmente poseen ambos tipos de tareas, así que haremos un análisis bastante general de cuáles métricas aplicar, destacando cuál métrica provee más información según el tipo de tarea que se vayan a utilizar. Para ello, utilizaremos el algoritmo de *Round – Robin* y calcularemos los valores de las métricas en base a los resultados arrojados por el algoritmo.

Hay características de los procesadores que en la práctica no pueden modificarse, ya que son limitaciones de *hardware*. Como el simulador nos permite modificar esos parámetros a gusto, decidimos dejarlos en un valor fijo e ir cambiando algún valor que sí podamos controlar en la práctica. Estos valores son:

- Tiempo de cambio de contexto, que dejaremos en 1 (un) ciclo de reloj.
- Tiempo de cambio de núcleo, que dejaremos fijo en 2 (dos) ciclos de reloj.

Nos queda entonces libres el parámetro de la cantidad de núcleos y el *quantum* de cada uno de ellos.

4.1.1. Fairness

Fairness o equanimidad es una métrica que permite analizar cuán justo es el algoritmo de *scheduling*. Un *scheduler* justo es aquel que reparte de forma bastante equitativa el acceso al procesador, de manera que no haya ningún proceso que monopolice el uso del procesador por sobre otros procesos. Cabe aclarar que no todos los procesos necesitan el procesador por igual: las tareas interactivas no lo usan intensivamente, ya que pasan mayor tiempo de su ejecución esperando datos de los dispositivos de entrada y salida. Tiene sentido entonces que esas tareas reciban un porcentaje menor del procesador.

El algoritmo de *Round – Robin* implementa el sistema de *quantum* para poder responder a la necesidad de que los procesos reciban un porcentaje similar del procesador. Nuestra intuición es que los resultados determinarán un algoritmo *fair* o ecúanime, veamos unos casos de prueba.

El lote de tareas contiene las siguientes tareas:

0. TaskBatch 10 8
1. TaskBatch 10 4
2. TaskBatch 10 4
3. @4:TaskBatch 10 8
4. @6:TaskBatch 10 2

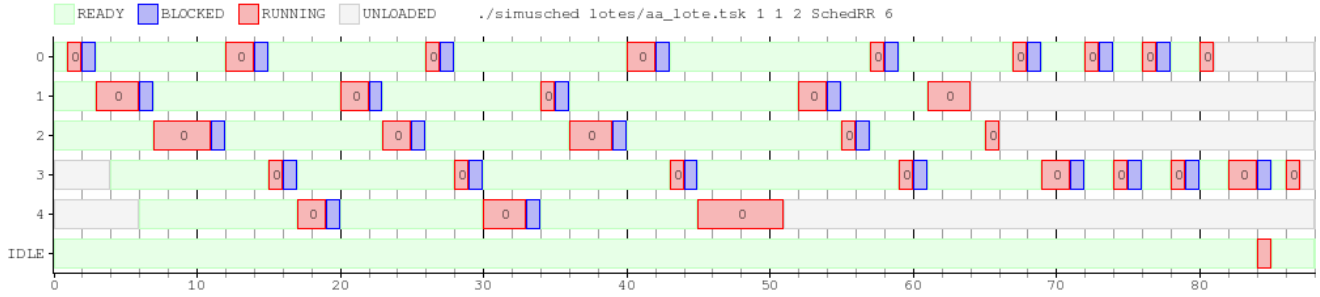


Figura 1: Diagrama de Gantt para el caso de 1 núcleo y 6 de quantum.

Tareas	Tarea 0	Tarea 1	Tarea 2	Tarea 3	Tarea 4	Total CPU
Quantum=1	10	10	10	10	10	110
Quantum=4	12,5	12,5	12,5	12,5	12,5	88
Quantum=6	12,64	12,64	12,64	12,64	12,64	87

Uso de CPU con un sólo núcleo (en porcentaje). El cambio de contexto para valores de quantum bajo utiliza mucho del procesador. Cuando el quantum es 1(unos), el procesador está 60 % haciendo tareas de mantenimiento. Esto ya fue observado anteriormente en este informe.

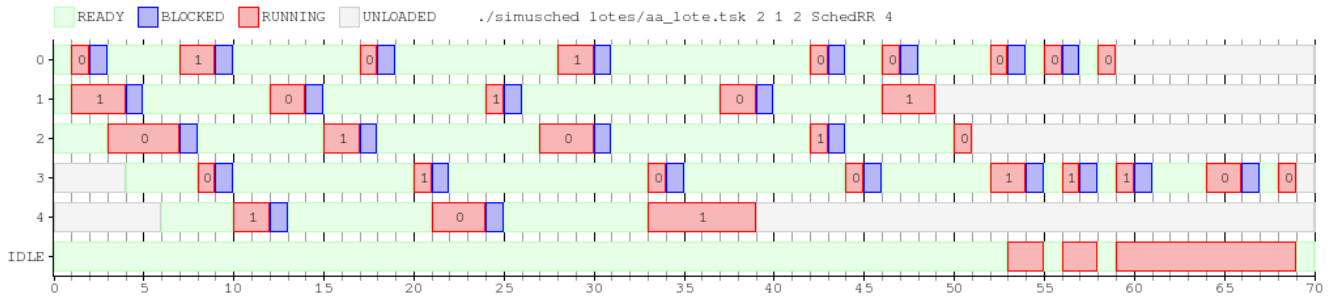


Figura 2: Diagrama de Gantt para el caso de 2 núcleos y 4 de quantum.

Tareas	Tarea 0	Tarea 1	Tarea 2	Tarea 3	Tarea 4	Total CPU
Quantum=1	16,67	16,67	16,67	16,67	16,7	66
Quantum=4	16	16	16	16	16	69
Quantum=6	16	16	16	16	16	69

Uso de CPU con un 2(dos) núcleos (en porcentaje).

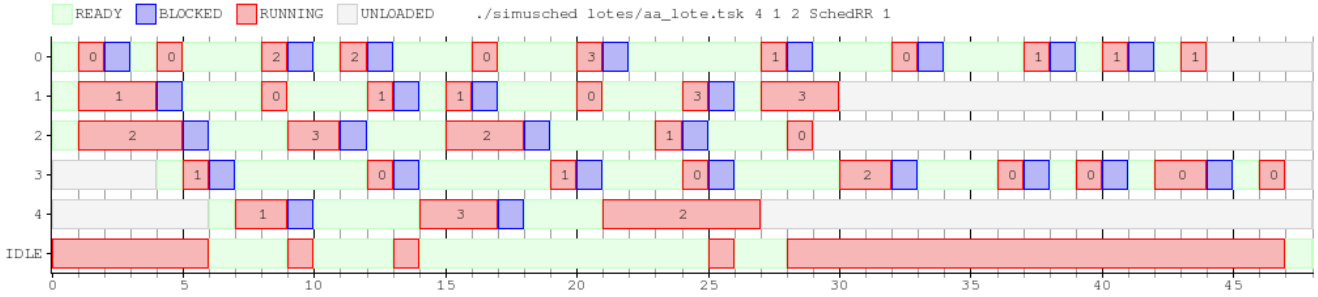


Figura 3: Diagrama de Gantt para el caso de 4 núcleos y 1 de quantum. Veamos que al aumentar la cantidad de núcleos aumenta el rendimiento de los núcleos, ya que casi siempre están ocupados. Es por esta razón también que la suma de porcentaje de uso de CPU da mayor a 100 porciento dado que en casi todos los momentos el procesador se encuentra ejecutando más de un proceso en simultáneo, y se duplica el rendimiento del uso de CPU.

Tareas	Tarea 0	Tarea 1	Tarea 2	Tarea 3	Tarea 4	Total CPU
Quantum=1	23	23	23	23	23	47
Quantum=4	23	23	23	23	23	47
Quantum=6	23	23	23	23	23	47

Uso de CPU con un 4(cuatro) núcleos (en porcentaje). Observar que las llamadas bloqueantes son mucho menores al quantum, así que aumentar el quantum no tiene efecto.

Por lo tanto nuestra intuición era correcta, el procesador se reparte de modo equitativa entre todas las tareas, y por lo tanto decimos que es *Round – Robin* es un algoritmo de *scheduling* justo o ecuánime.

4.1.2. Response time

Otra métrica conocida es la del *Response time* o tiempo de respuesta, que se define como el tiempo que tarda una tarea en devolver datos después de una llamada bloqueante para acceder a recursos de entrada y salida. Esta medida también es conocida como latencia e impacta directamente sobre la experiencia del usuario. Una latencia elevada da la impresión al usuario de que el sistema anda lento o se cuelga. Es una métrica principalmente aplicable a las tareas interactivas, ya que realizan varias de estas llamadas. Veamos que *Round – Robin* puede no ser algoritmo con buen *response time*, ya que este depende de la cantidad de tareas que se estén ejecutando en un mismo momento.

Como las tareas del tipo TaskBatch realizan varias llamadas bloqueantes, entonces hay varios valores de *response time*. Por lo tanto, haremos el promedio con todos ellos. Veamos nuevamente los gráficos.

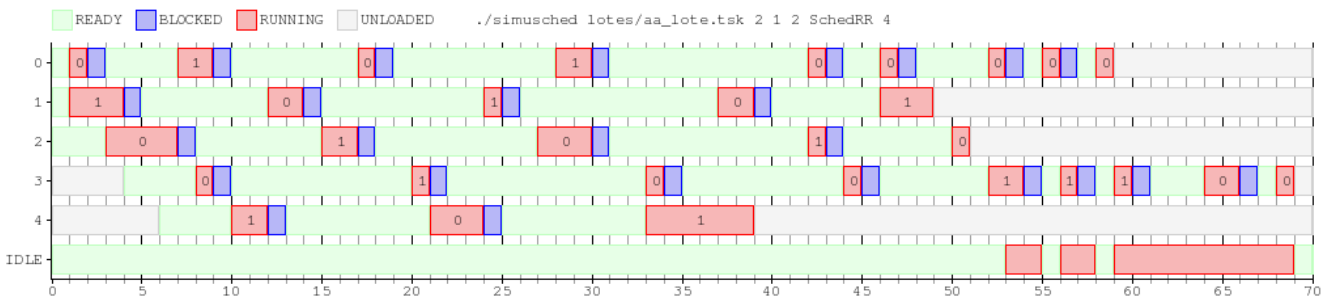


Figura 4: Diagrama de Gantt para el caso de 2 núcleos y 4 de quantum. El espacio verde entre el hilo de ejecución de una tarea es el *response time*, así que visualmente nos podemos dar una idea de sus valores.

Tareas	<i>Response times</i>	<i>Response time</i> promedio
Tarea 0	[5, 8, 10, 12, 3, 5, 2, 2]	5.875
Tarea 1	[8, 10, 12, 7]	9.25
Tarea 2	[8, 10, 12, 7]	9.25
Tarea 3	[11, 12, 10, 7, 2, 2, 4, 2]	6,25
Tarea 4	[9, 9]	9

Los valores de *response time* del lote de tareas. Nótese que al principio el sistema tiene muchos procesos por ejecutar, y eso hace que el *response time* sea bastante mayor durante ese tiempo, y esto se reduce al final, cuando algunas tareas ya terminaron su ejecución.

El problema principal de *Round – Robin* con respecto al *response time* es su implementación de cola de procesos; una tarea que realiza una llamada bloqueante debe esperar que le toque volver a tocar usar el procesador. Si hay muchas tareas entonces el proceso tardará mucho en devolver resultados al usuario, y dará la apariencia de "cuelgue" de sistema.

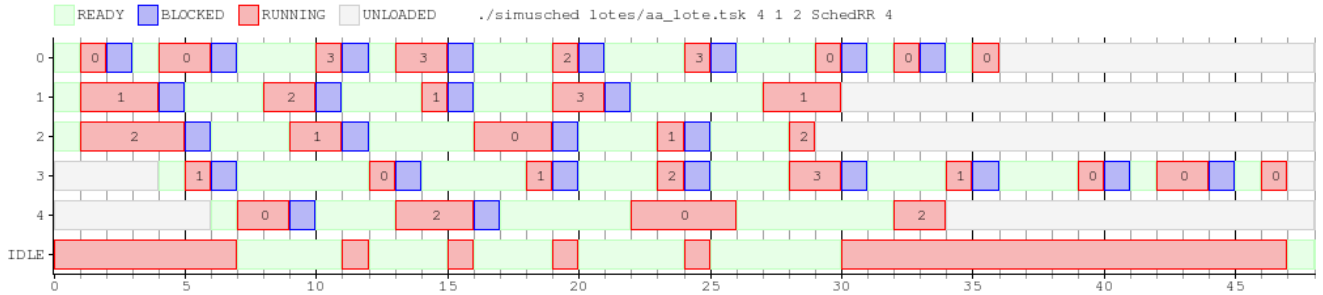


Figura 5: Diagrama de Gantt para el caso de 4 núcleos y 4 de quantum.

Tareas	<i>Response times</i>	<i>Response time</i> promedio
Tarea 0	[2, 4, 2, 4, 4, 4, 2, 2]	3
Tarea 1	[4, 4, 4, 6]	4.5
Tarea 2	[4, 5, 4, 4]	4.25
Tarea 3	[6, 5, 4, 4, 4, 4, 2, 2]	3.875
Tarea 4	[4, 6]	5.0

Como se puede observar, aumentar la cantidad de núcleos contribuye a reducir los *response times* porque paraleliza los hilos de ejecución y los procesos pueden ser levantados por otro núcleo para continuar con su trabajo. Esto no es gratis, se debe pagar un costo de migración de núcleo, pero al fin y al cabo ese gasto es totalmente justificado.

Concluimos entonces que si bien en principio *Round – Robin* no tiene un buen rendimiento en lo que respecta a *response time*, si se disponen de varios núcleos entonces se comporta muchísimo mejor.

Referencias

- [1] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson, 4th edition, 2013.
- [2] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. *The 1994 Operating Systems Design and Implementation conference (OSDI '94)*, 1994. https://www.usenix.org/legacy/publications/library/proceedings/osdi/full_papers/waldspurger.pdf.