Trabajo Práctico de Sistemas Operativos

26 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

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 cómputos 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, .., n_k)
2:     for i ← [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 ← 0, cant_bloqueos do
4:         random_number ← 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:

$$\begin{aligned} \texttt{rand}() \in [0..RAND\_MAX] &\implies \texttt{rand}() \textbf{ mód } (bmax - bmin + 1) \in [0..bmax - bmin] \\ &\implies \texttt{rand}() \textbf{ mód } (bmax - bmin + 1) + bmin \in [bmin..bmax] \end{aligned}$$

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 ← elegirMomentosParaBloquear(total_cpu, cant_bloqueos)  ▷ [Nota 2]
3:     for i ← [1..total_cpu] do                                                      ▷ 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*

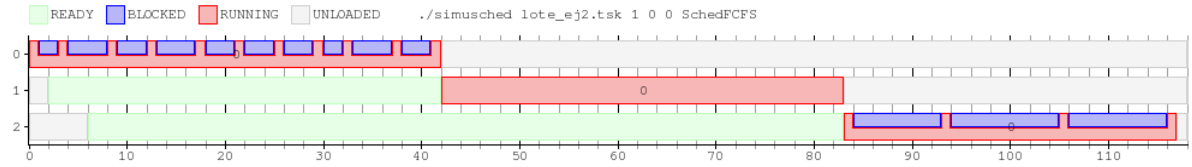
3.1. First-Come First-Served (FCFS)

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

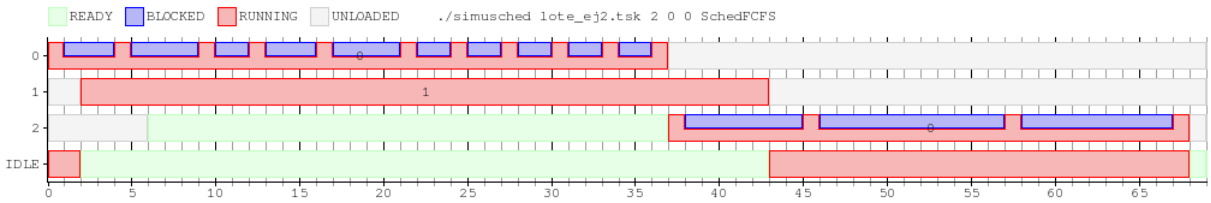
```
TaskConsola 10 2 4
@2:
TaskCPU 40
@6:
TaskConsola 3 6 11
```

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.

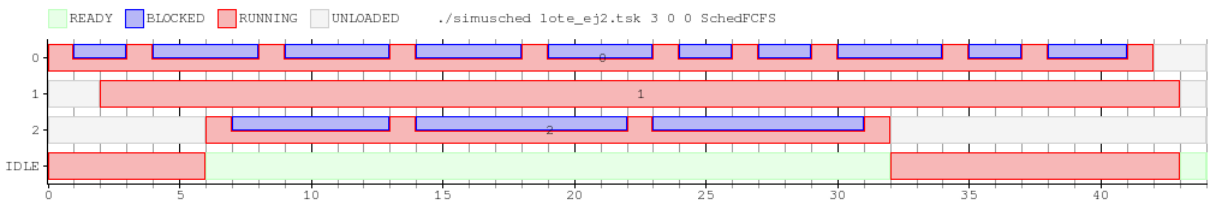
A continuación se muestran los gráficos para 1, 2 y 3 núcleos, usando SchedFCFS para el lote de tareas del cuadro.



2



3



Obs: Los parámetros de costo de cambio de contexto y migración fueron seteados en 0 ya que son irrelevantes para este scheduller. 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 cántidad de núcleos, mejor rendimiento. Esto se debe a que FCFS no soporta múltitarea por sí sólo (ya que núnca desaloja a las tareas), sino que necesita varios núcleos para lograrlo.

Las desventajas de FCFS son varias:

- Como ya dijimos, no soporta múltitarea.
- 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.

En nuestro caso usamos las siguientes estructuras de datos:

Cola de procesos 'Ready'

Algorithm 6 Round-Robin

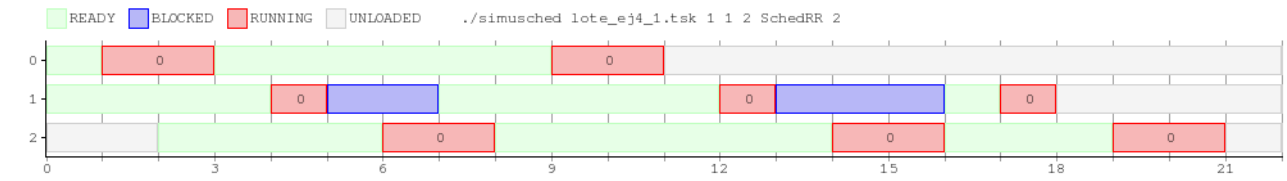
```
1: procedure LOAD(pid)
2:     encolo la nueva tarea en readyTasks
3: end procedure
4: procedure UNBLOCK(pid)
5:     encolo la tarea desbloqueada en readyTasks
6: end procedure
7: procedure TICK(cpu, motivo)
8:     if el cpu esta ejecutando IDLE then
9:         llamo a NEXT(cpu) para obtener la próxima 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
```

Para las simulaciones mantuvimos los siguientes costos:

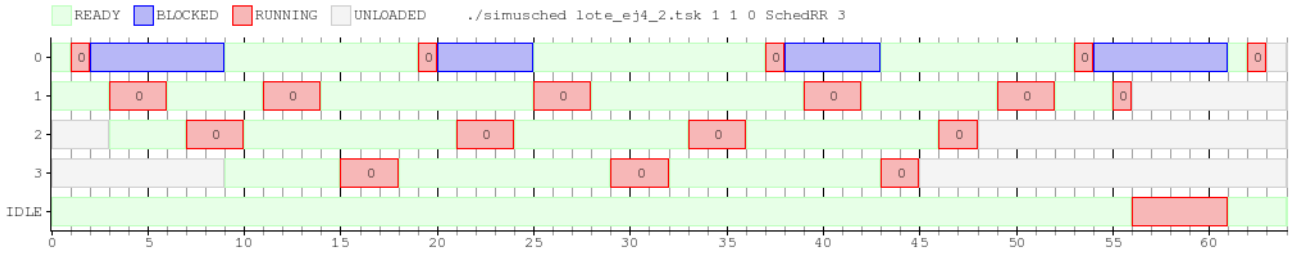
costo de cambio de contexto = 1 : En un CPU real se deben intercambiar estructuras de datos que contienen información de los procesos antes de poder correr la nueva tarea.

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.



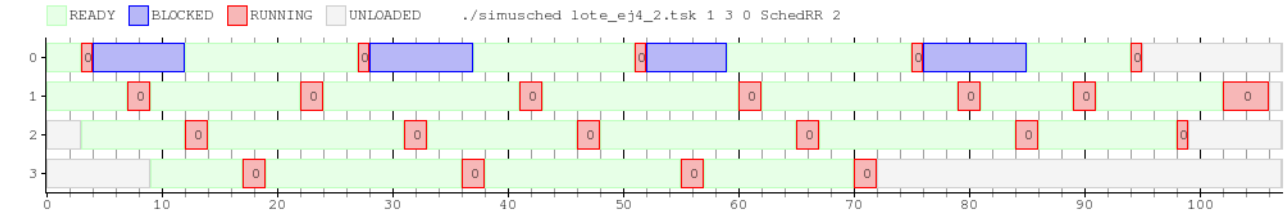
Se puede observar que no es un algoritmo FCFS porque, en principio, hace desalojo de tareas cada cierto quantum. Además las tareas que se bloquean son desalojadas de inmediato, sin encolarlas en ReadyTasks hasta que se desbloqueen.



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 esté 'más libre', es decir, 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 schedulling, 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.

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 alternandose entre largos y cortos entonces:

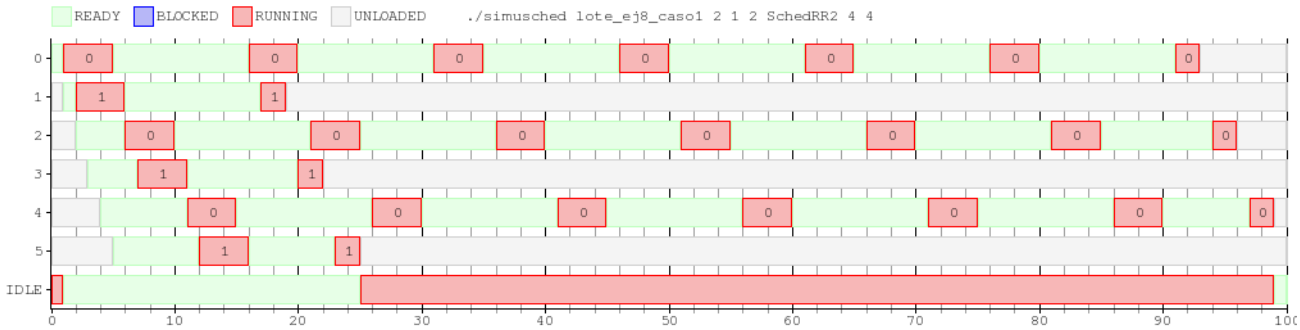
- Al primer proceso (largo) le es asignado el núcleo 0.
- Al segundo proceso (corto) se le asigna el núcleo 1.
- Al tercer proceso (largo) le es asignado el núcleo 0.

  ... y continúa así.

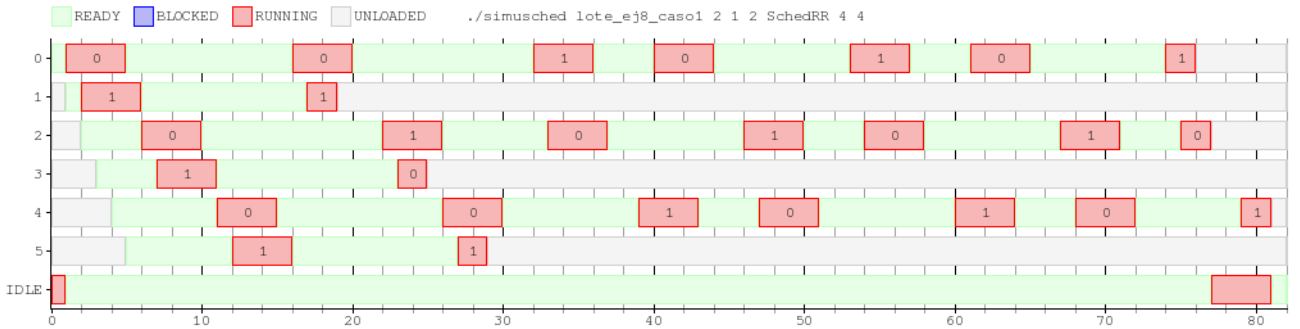
Con lo cuál, luego de la carga del último proceso quedan distribuidos de la siguiente manera:
Núcleo 0: 3 procesos largos. Núcleo 1: 3 procesos cortos.

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)



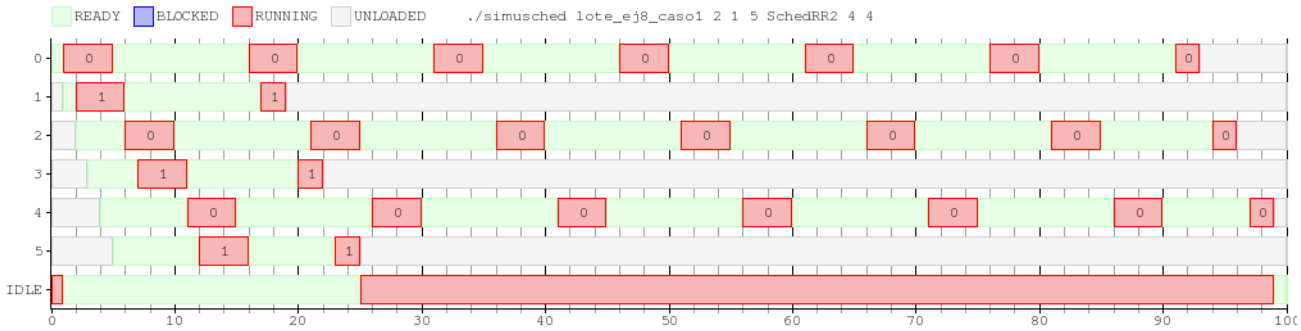
Este escenario también se podría dar si en vez de tareas cortas tenemos tareas que realizan llamadas bloqueantes. Entonces en el núcleo 0 hay tareas usando CPU y el núcleo 1 queda esperando que algún proceso se desbloquee.

Pro - caso: Migración costosa

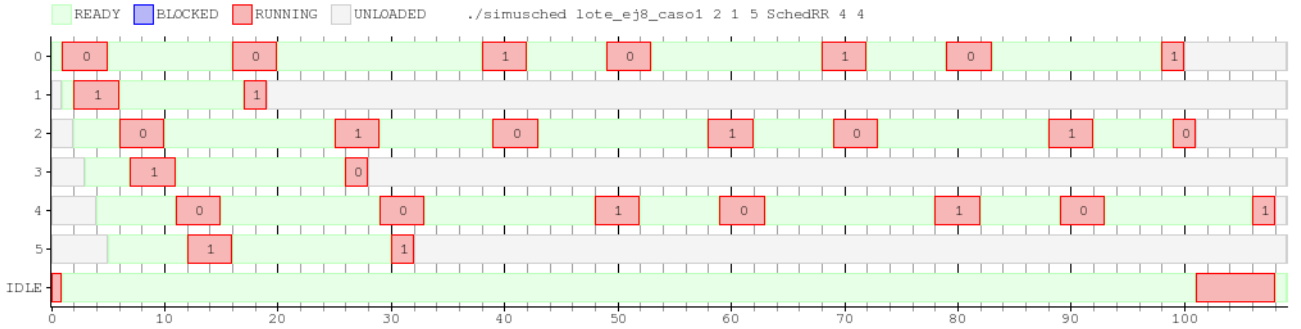
Una de las ventajas de RR2 queda expuesta si el costo de la migración es grande. Si los procesos cambian mucho entre núcleos puede que en el caso de RR el procesador pase más tiempo haciendo tareas de mantenimiento que efectivamente corriendo procesos.

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*[?] 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*, puediendo 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. Está 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 $\{\mathrm{t}_i\}_{1 \leq i \leq n}$ 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.