DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Sistemas Operativos

Trabajo práctico 1

Scheduling

Resumen

Integrante	LU	Correo electrónico
Acosta, Javier Sebastian	338/11	acostajavier.ajs@gmail.com
Gomez, Fernando Nahuel	695/11	fernando.gmz12@gmail.com
Russo, Christian Sebastián	679/10	christian.russo@gmail.com

Palabras claves:

Scheduling

Índice

1. Pregunta 1 **4**
1.1. Enunciado del Problema: . . . 4
1.2. Pseudocodigo . . . 4
1.3. Explicacion . . . 4

2. Pregunta 2 **4**
2.1. Enunciado del Problema: . . . 4
2.2. Lote . . . 4
2.3. Resultados . . . 5

3. Pregunta 3 **6**
3.1. Enunciado del Problema: . . . 6
3.2. Pseudocodigo . . . 6
3.3. Explicación de la implementacion . . . 7

4. Pregunta 4 **8**
4.1. Enunciado del Problema: . . . 8
4.2. Experimentos . . . 8
4.2.1. Primer experimento: . . . 8
4.2.2. Segundo experimento: . . . 9

5. Pregunta 5 **10**
5.1. Enunciado del Problema: . . . 10
5.2. Explicacion de la implementacion . . . 10
5.3. Pseudocodigo . . . 11
5.4. Optimizaciones propuestas por el articulo . . . 13

6. Pregunta 6 **13**
6.1. Enunciado del Problema: . . . 13
6.2. Pseudocodigo . . . 14
6.3. Explicación de la implementacion . . . 14

7. Pregunta 7 **14**
7.1. Enunciado del Problema: . . . 14
7.2. Experimento con 1 core . . . 15
7.3. Experimento con 2 cores . . . 17
7.4. Experimento con 4 cores . . . 20

8. Pregunta 8 **22**

8.1. Enunciado del Problema: . . . 22
8.2. Explicación de la implementacion . . . 22
8.3. Comparación entre los algoritmos de Round Robin . . . 22
8.3.1. Primer experimento . . . 23
8.3.2. Segundo experimento . . . 23
8.3.3. Conclusión . . . 24

9. Pregunta 9 **25**
9.1. Enunciado del Problema: . . . 25
9.1.1. Introducción . . . 25
9.1.2. Experimentación con un núcleo . . . 26
9.1.3. Experimentación con dos núcleos . . . 28
9.1.4. Conclusión . . . 30

10.Pregunta 10 **30**
10.1. Enunciado del Problema: . . . 30
10.2. Solución . . . 30

1. Pregunta 1

1.1. Enunciado del Problema:

Programar un tipo de tarea TaskConsola, que simulará una tarea interactiva. La tarea debe realizar n llamadas bloqueantes, cada una de una duracion al azar1 entre bmin y bmax (inclusive). La tarea debe recibir tres parametros: n, bmin y bmax (en ese orden) que serian interpretados como los tres elementos del vector de enteros que recibe la funciion.

1.2. Pseudocodigo

Algorithm 1 TaskConsola(pid, params)

```
n = params[0]
bmin = params[1]
bmax = params[2]
for i = 0 hasta n do
  uso_IO(pid, bmin + rand() % (bmax - bmin +1))
end for
```

1.3. Explicacion

Para realizar una llamada **bloqueante** hacemos uso de la funcion uso_IO, para utilizar la entrada y salida. Hacemos un ciclo que ejecuta **n** veces la funcion con el parametro de duracion al azar en el intervalo dado. Tambien registramos nuestra nueva tarea en task_init().

2. Pregunta 2

2.1. Enunciado del Problema:

Escribir un lote de 3 tareas distintas: una intensiva en CPU y las otras dos de tipo interactivo (TaskConsola). Ejecutar y graficar la simulacion usando el algoritmo FCFS para 1, 2 y 3 nucleos.

2.2. Lote

```
@2
TaskCPU 10
@5
TaskConsola 9 1 7
@3
TaskConsola 4 1 5
```

2.3. Resultados

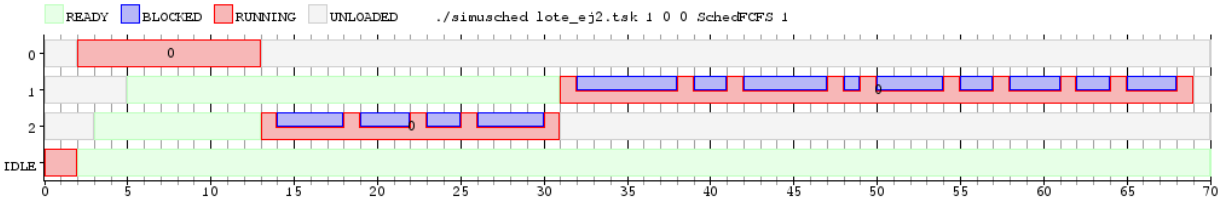

Figura 1: FCFS con 1 núcleo

Conclusiones: Se puede ver claramente que se ejecutan las tareas de forma FCFS, es decir primero se ejecuta el proceso 0 ya que es el primero en llegar (2 unidades de tiempo) una vez finalizado se pasa la ejecucion al proceso 2 (que llego en 3 unidades de tiempo), este ejecuta las 4 llamadas bloquenates y finalmente se pasa la ejecucion al proceso 1 que fue el ultimo en llegar (5 unidades de tiempo) ejecutando las 9 llamadas bloquentes. Tambien se puede ver a simple vista que las llamadas bloqueantes, respetan el bmin y bmax

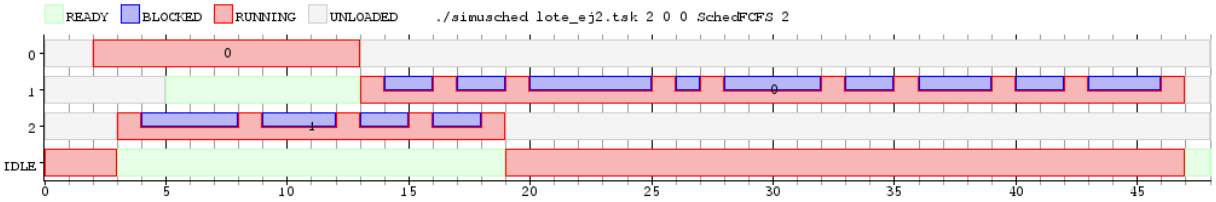

Figura 2: FCFS con 2 núcleos

Conclusiones: En este experimento se puede ver que el funcionamiento es similar al experimento anterior, la única diferencia es que mientras se ejecuta el proceso 0, es decir cuando llega el proceso 2, al haber dos núcleos este no espera a la finalizacion del proceso 0 sino que arranca directamente su ejecución en el otro núcleo. Finalmente el proceso 1 (el restante) se ejecuta en el núcleo 0 cuando el proceso 0 (que es el primero en terminar) termina (es decir, se libera un núcleo).

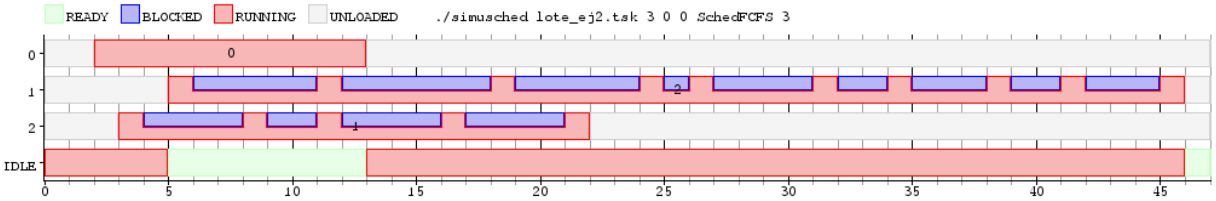

Figura 3: FCFS con 3 núcleos

Conclusiones: Idéntico al caso anterior, pero con diferencia de que ahora tenemos tres núcleos y, por lo tanto, al haber solo 3 procesos, ninguna tiene que esperar a que otro termine, ya que cada uno puede correr en un núcleo distinto. Es por ese motivo que cada proceso se ejecuta en el instante en que llega.

3. Pregunta 3

3.1. Enunciado del Problema:

Completar la implementacion del scheduler Round-Robin implementando los metodos de la clase SchedRR en los archivos sched rr.cpp y sched rr.h. La implementacion recibe como primer parametro la cantidad de nucleos y a continuacion los valores de sus respectivos quantums. Debe utilizar una unica cola global, permitiendo asi la migracion de procesos entre nucleos.

3.2. Pseudocodigo

Para este ejercicio manejamos una cola q en la cual tenemos todos los procesos ready.

Algorithm 2 SchedRR(Vector<int>argn)

```
cantCores = argn[0]
cpuQuantum = Vector<int>de tamaño cantCores y cero en todas las posiciones.
quantumActual = Vector<int>de tamaño cantCores y cero en todas las posiciones.
for (i = 0 hasta cantCores) do
    quantumActual[i] = argn[i+2]
    cpuQuantum[i] = argn[i+2]
end for
```

Algorithm 3 load(pid)

```
q.push(pid)
```

Algorithm 4 unblock(pid)

```
q.push(pid)
```

Algorithm 5 tick(cpu , Motivo m)

```
if (m == EXIT o m == BLOCK) then
  if (q.vacia()) then
    Devolver run_idle_task(cpu)
  else
    quantumActual[cpu] = cpuQuantum[cpu]
    prox = q.front()
    q.pop()
    Devolver prox
  end if
else
  if (current_pid(cpu) == IDLE_TASK) then
    if (q.vacia()) then
      Devolver devolver run_idle_task(cpu)
    else
      Devolver proximo_proceso(cpu)
    end if
  else
    quantumActual[cpu] = quantumActual[cpu] - 1
    if (quantumActual[cpu] == 0) then
      if (q.vacia()) then
        quantumActual[cpu] = cpuQuantum[cpu]
        Devolver current_pid(cpu)
      else
        Devolver proximo_proceso(cpu)
      end if
    else
      Devolver current_pid(cpu)
    end if
  end if
end if
```

Algorithm 6 proximo_proceso(cpu)

```
quantumActual[cpu] = cpuQuantum[cpu]
prox = q.front()
q.pop()
if (current_pid(cpu) != -1) then
  cola.push(current_pid(cpu))
end if
Devolver prox
```

Algorithm 7 run_idle_task(cpu)

```
quantumActual[cpu] = cpuQuantum[cpu]
Devolver IDLE_TASK
```

3.3. Explicación de la implementacion

Para implementar el scheduler Round Robin utilizamos una cola de procesos *ready*. Cuando un núcleo del procesador queda libre *(ya sea por que el proceso ejecutándose termino su quantum,*

se bloqueo o termino su ejecución) hacemos que el próximo proceso a ejecutarse en ese núcleo sea el primer proceso en la cola de procesos *ready (en caso de no haber ninguno, el procesador queda idle).*

Cuando un proceso se desbloquea es colocado al final de la cola de procesos *ready* a la espera de que algún núcleo del procesador quede libre para ejecutarse.

Si un núcleo esta idle *(es decir, ejecutando la tarea idle)* y no hay ningún proceso *ready* para ejecutar, simplemente se queda a la espera de que algún proceso este *ready.*

4. Pregunta 4

4.1. Enunciado del Problema:

Disenar uno o mas lotes de tareas para ejecutar con el algoritmo del ejercicio anterior. Graficar las simulaciones y comentarlas, justificando brevemente por que el compor- tamiento observado es efectivamente el esperable de un algoritmo Round-Robin.

4.2. Experimentos

En los experimentos siguiente fijaremos el quantum en un ciclo de reloj, el costo de cambio de contexto también en un ciclo y el costo de migrar entre núcleos en dos ciclos de reloj.

4.2.1. Primer experimento:

En este primer experimento utilizaremos un lote de tres procesos que utilizan intensivamente el CPU:

@1
TaskCPU 5
@3
TaskCPU 4
@4
TaskCPU 4

Solo utilizamos un núcleo de procesamiento, ya que lo que nos interesa mostrar en este experimento es que los procesos se ejecutan de la forma en que Round Robin lo especifica. A continuación el gráfico resultante:

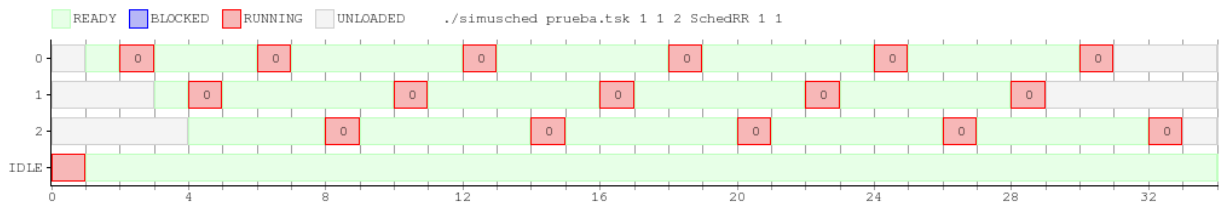

Figura 4: Experimentación de Round Robin

Conclusiones:

Los procesos se ejecutan de forma alternada, a medida que se van agregando a la lista de procesos *ready*, y solo durante el quantum especificado.

4.2.2. Segundo experimento:

En este segundo experimento utilizamos un lote con seis procesos. Tres procesos que utilizan intensivamente el CPU y tres procesos que realizan operaciones de E/S. El lote utilizado es el siguiente:

```
@1
TaskCPU 5
@2
TaskConsola 3 3 3
@3
TaskCPU 4
@4
TaskCPU 4
TaskConsola 2 4 4
@5
TaskConsola 1 5 5
```

Fijamos en dos (2) la cantidad de núcleos del procesador, ya que con un solo núcleo no es posible observar que el scheduler realmente realiza migraciones de procesos entre núcleos, y con mas de dos núcleos no se obtiene ninguna información del scheduler que no se obtenga con dos núcleos.

A continuación, el gráfico resultante de ejecutar el lote de procesos con el scheduler Round-Robin:

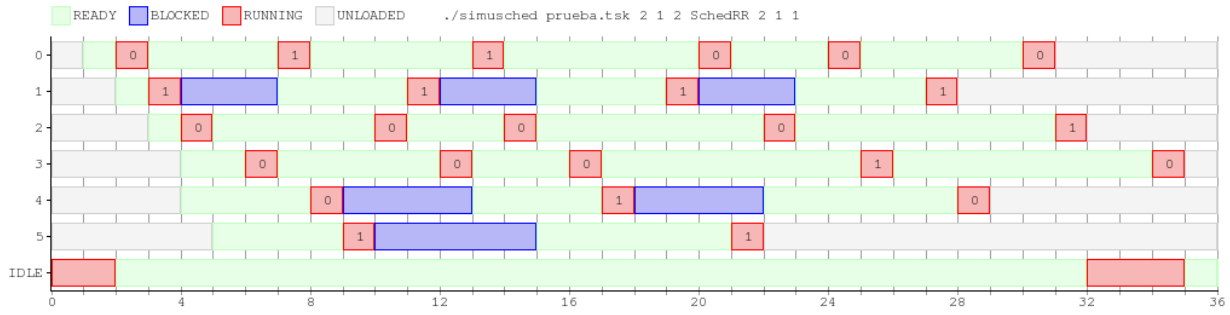

Figura 5: Experimentación de Round Robin

Conclusiones:

Como se puede ver en el gráfico, los procesos se ejecutan en el primer núcleo que este libre independientemente de si se ejecuto previamente en ese núcleo o no, permitiendo así la migración de procesos entre núcleos. También se puede ver que los procesos solo se ejecutan durante el quantum especificado y luego son desalojados para permitir a otro proceso utilizar el procesador. Cuando un proceso se desbloquea o consume todo su quantum es añadido al final de la cola de

procesos *ready* lo cual, en muchos casos, altera el orden en el cual se ejecutan los procesos, es decir, los procesos no siempre se ejecutan en el orden en el cual llegaron.

5. Pregunta 5

5.1. Enunciado del Problema:

A partir del articulo

- Waldspurger, C.A. and Weihl, W.E., Lottery scheduling: Flexible proportional-share resource management. Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation -1994.

disenar e implementar un scheduler basado en el esquema de loteria. El constructor de la clase SchedLottery debe recibir dos parametros: el quantum y la semilla de la secuencia pseudoaleatoria (en ese orden). Interesa implementar al menos la idea basica del algoritmo y la optimizacion de tickets compensatorios (compensation tickets). Otras optimizaciones y refinamientos que propone el articulo serian opcionales siempre que, en cada caso, se explique brevemente por que la optimizacion no se considero relevante a los efectos de este trabajo.

5.2. Explicacion de la implementacion

Para implementar [1] esta política de scheduling utilizamos:

- Un entero para llevar la cuenta de cuantos tickets hay en total entre todos los procesos *ready*.
- Un diccionario donde la clave es el *pid* del proceso y el valor es la cantidad de tickets que tiene dicho proceso. Solo los procesos *ready* están definidos en esta estructura.
- Otro diccionario indexado por el *pid* del proceso que permite sabe, en caso de que el proceso se haya bloqueado, cuantos ciclos de su *quantum* utilizo.

Lo primero que hacemos es utilizar la semilla pasada como parámetro al constructor del scheduler para inicializar el generador de números pseudo-aleatorio con la función: *srand(semilla)*.

Cada vez que un proceso se añade al scheduler, le otorgamos un valor constante de tickets *(1000 tickets)*. De esta forma, todos los procesos tienen, a priori, la misma probabilidad de ganar el uso del procesador. Cuando un proceso gana el uso procesador, este lo tiene asignado durante un quantum *(parámetro del scheduler)* y luego es desalojado.

Cuando un proceso se bloquea determinamos la cantidad **C** de ciclos del quantum asignado que utilizo, es decir, restar al Quantum del CPU la cantidad de ciclos utilizados por el proceso actual en el núcleo actual *(este valor lo tenemos en un vector para cada núcleo)*. Luego, cuando se desbloquea le otorgamos el *compensation ticket* de forma que su cantidad total de tickets pasa a ser: $tickets * quantum \ / \ ciclosUtilizados$. Cuando un proceso con un *compensation ticket* gana el uso del procesador, pierde dicho ticket.

Para determinar cual es el proceso que gana el uso del procesador, llevamos a cabo una lotería entre todos los procesos *ready*. Es decir, elegimos un numero al azar entre cero y la cantidad total de tickets entre todos los procesos *ready*, este sera el ticket ganador. Luego, recorremos el diccionario que contiene a todos los procesos *ready* sumando en un contador la cantidad de tickets de cada proceso hasta encontrar el proceso que hace que el valor de dicha sumatoria sea mayor o igual que el valor del ticket ganador. Este sera el proceso que ganara el uso del procesador,

[1] La implementacion de este scheduler se puede encontrar en el archivo *sched_lottery.cpp*.

entonces restamos los tickets de este proceso de la cantidad de tickets total y lo eliminamos de la estructura que contiene a todos los procesos *ready*.

5.3. Pseudocodigo

Algorithm 8 SchedLottert(vector<int>argn)

BASE_TICKETS = 1000
cores = argn[0]
cpuQuantum = argn[1]
semilla = argn[2]
srand(semilla)
quantumActual = Vector<int>de tamaño cores y cpuQuantum en cada posición;
procesos = Diccionario donde la clave es el pid del proceso y el valor asociado es la cantidad de tickets del proceso
bloqueados = Diccionario donde la clave es el pid del proceso y el valor asociado es la cantidad de ciclos de su quantum que utilizo antes de bloquearse;
ticketsTotales = 0

Algorithm 9 load(pid)

load(pid, 0)

Algorithm 10 load(pid,deadline)

procesos[pid] = BASE_TICKETS
ticketsTotales = ticketsTotales + BASE_TICKETS

Algorithm 11 unblock(pid)

procesos[pid] = (BASE_TICKETS * cpuQuantum / bloqueados[pid])
ticketsTotales = ticketsTotales + procesos[pid]

Algorithm 12 tick(cpu , Motivo m)

```
if (m == EXIT) then
  if (procesos.vacio()) then
    Devolver run_idle_task(cpu)
  else
    Devolver loteria(-1, cpu)
  end if
  if (m == BLOCK) then
    bloqueados[current_pid(cpu)] = cpuQuantum - quantumActual[cpu]
    if (procesos.vacio()) then
      Devolver run_idle_task(cpu)
    else
      Devolver loteria(-1, cpu)
    end if
  else
    quantumActual[cpu] = quantumActual[cpu] -1
    if (quantumActual[cpu] == 0) then
      if (procesos.vacio()) then
        Devolver current_pid(cpu)
      else
        Devolver loteria(current_pid(cpu), cpu)
      end if
    else
      Devolver current_pid(cpu)
    end if
  end if
end if
```

Algorithm 13 loteria(pid, cpu)

```
quantumActual[cpu] = cpuQuantum
if (pid != -1) then
  procesos[pid] = BASE_TICKETS
  ticketsTotales = ticketsTotales + BASE_TICKETS
end if
ticketGanador = rand() % (ticketsTotales + 1)
procesoGanador = El pid de algún proceso
sumatoria = 0
for (Proceso p en procesos) do
  sumatoria = sumatoria + p.tickets
  if (sumatoria >= ticketGanador) then
    procesoGanador = p.pid
    Salir del for
  end if
end for
ticketsTotales = ticketsTotales - procesos[procesoGanador]
procesos.erase(procesoGanador)
Devolver procesoGanador
```

Algorithm 14 run_idle_task(cpu)

```
quantumActual[cpu] = cpuQuantum
devolver IDLE_TASK
```

5.4. Optimizaciones propuestas por el articulo

En el articulo se proponen algunas optimizaciones, a continuación nombramos cada una de ellas y el motivo por el cual fue o no fue implementada:

- **Transferencia de tickets**: Esta optimizacion no la implementamos ya que,en el simulador utilizado, los procesos solo se bloquean a la espera de la E/S. Nunca se bloquean a la espera de otros procesos.
- **Inflación de tickets**: No nos pareció correcto que un proceso tenga la capacidad de otorgarse todos los tickets que desee, ya que de esta forma un proceso podría monopolizar el procesador.
- **Tickets de compensación**: Fue implementado.
- **Ticket currencies**: Dado que los procesos, en el simulador, no tienen procesos hijos ni hay varios usuarios ejecutando procesos, no nos pareció necesario implementarlo. Ademas, seria imposible testear dicha implementacion.
- **Ordenar en forma decreciente la lista de procesos ready según la cantidad de tickets del proceso**: Esta es una optimizacion asintotica. Realmente harían falta muchos procesos para que tuviera un impacto notable, y por este motivo decidimos no implementarla.

6. Pregunta 6

6.1. Enunciado del Problema:

Programar un tipo de tarea TaskBatch que reciba dos parametros: total cpu y cant bloqueos. Una tarea de este tipo debera realizar cant bloqueos llamadas bloqueantes, en momentos elegidos pseudoaleatoriamente. En cada tal ocasion, la tarea debera permanecer bloqueada durante exactamente un (1) ciclo de reloj. El tiempo de CPU total que utilice una tarea TaskBatch deberia ser de total cpu ciclos de reloj (incluyendo el tiempo utilizado para lanzar las llamadas bloqueantes; no asi el tiempo en que la tarea permanezca bloqueada).

6.2. Pseudocodigo

Algorithm 15 TaskBatch(pid, params)

```
ciclosTotales = params[0]
entradasBloqueantes = params[1]
vector<int>ocupado(ciclosTotales,0)
for i = 0 hasta entradasBloqueantes do
  momento = rand() % ciclosTotales
  while ocupado[momento] do
    momento = rand() % ciclosTotales
  end while
  ocupado[momento] = 1
end for
for i = 0 hasta ciclosTotales do
  if (ocupado[i]) then
    uso_IO(pid, 1)
  else
    uso_CPU(pid, 1)
  end if
end for
```

6.3. Explicación de la implementacion

Se crea un vector de tamaño *ciclosTotales* (es la cantidad total de ciclos de uso de CPU de la tarea) inicializado en cero. Lo interpretaremos como un vector de booleanos, donde si la posición i esta en 1, es porque en el momento i se hace una llamada bloqueante y cero en caso de que sea un ciclo solo de uso de CPU. Al principio en todas los momentos se esta usando solo CPU.

Luego se itera *entradasBloqueantes* (es la cantidad de llamadas bloqueantes en la tarea) veces y se elije un momento al azar entre 0 y *ciclosTotales*. Si ese momento ya estaba ocupado ($ocupado[momento] = TRUE$) entonces se vuelve a elegir al azar para tomar otro momento. Luego se setea en uno indicando que habrá una llamada bloqueante en ese momento.

Luego se recorre el vector y se ejecuta la tarea. Si el $ocupado[momento] = TRUE$ entonces se ejecuta un ciclo de uso de CPU y una llamada bloqueante, sino se ejecuta un solo ciclo de CPU.

7. Pregunta 7

7.1. Enunciado del Problema:

Disenar un lote de tareas TaskBatch, todas ellas con igual uso de CPU, pero con diversas cantidades de bloqueos. Simular este lote utilizando el algoritmo SchedRR y una variedad apropiada de valores de quantum. Mantener fijo en un (1) ciclo de reloj el costo de cambio de contexto y dos (2) ciclos el de migracion. Deben variar la cantidad de nucleos de procesamiento. Para cada una de las metricas elegidas, concluir cual es el valor optimo de quantum a los efectos de dicha metrica.

Elegimos 2 métricas para analizar las tareas TaskBatch: Turnarround y Waiting time.

Turnarround es una métrica que analiza cuanto tarda un proceso en terminar (en promedio), se puede usar para analizar el tiempo de ejecución de un programa en promedio.

Waiting time analiza el tiempo en que un proceso esta en la cola de listos en su vida, se usa

para analizar lo que tiene que esperar un proceso para ejecutarse y también el tiempo en que no esta haciendo nada.

Usamos el scheduling Round Robin con costo 1 en cambio de contexto y 2 en migración entre núcleos.

Para los gráficos, realizamos un lote de 5 tareas batch dejando fijo la cantidad de ciclos totales para las tareas en 10 y variando entre 1,2,3,4 y 5 llamadas bloqueantes para cada proceso (en ese orden).

En los primeros graficos dejamos fijo la cantidad de cores en 1 y variamos el quantum en cada uno en 1,3,6,9,10. Calculamos para cada uno Turnarround y Waiting time.

El Turnarround lo calculamos dividiendo $\frac{ciclosP_0+ciclosP_1+ciclosP_2+ciclosP_3+ciclosP_4}{5}$. Eso nos devuelve la cantidad de ciclos que le toma a un proceso en ejecutarse en promedio.

El Waiting time se calcula tomando el promedio de tiempo de espera de cada proceso, cuyo valor se calcula sumando los ciclos en donde el proceso esta en la cola de listos.

7.2. Experimento con 1 core

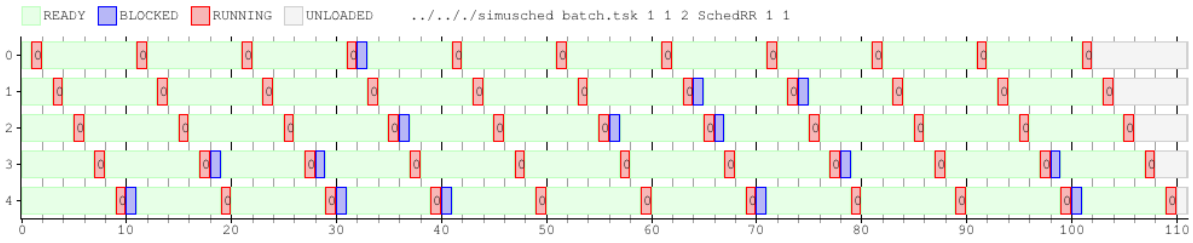

Figura 6: Round Robin con quantum 1

Turnarround: 105 ciclos

Waiting time: 92 ciclos

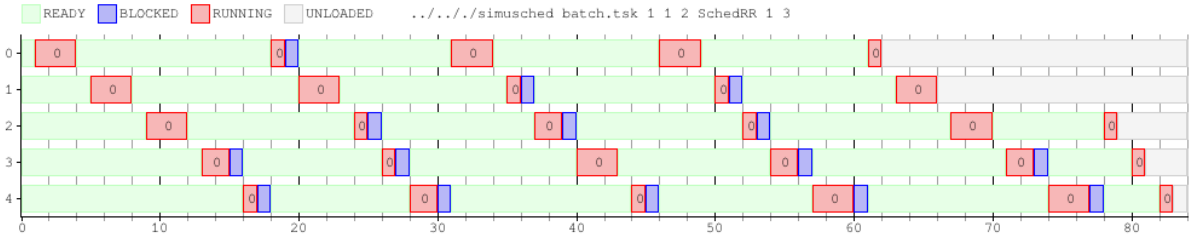

Figura 7: Round Robin con quantum 3

Turnarround: 74.2 ciclos

Waiting time: 60.2 ciclos

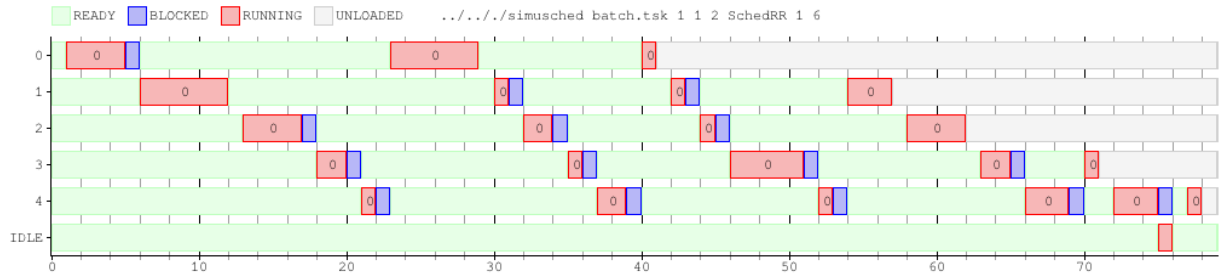

Figura 8: Round Robin con quantum 6

Turnarround: 62.8 ciclos

Waiting time: 47.8 ciclos

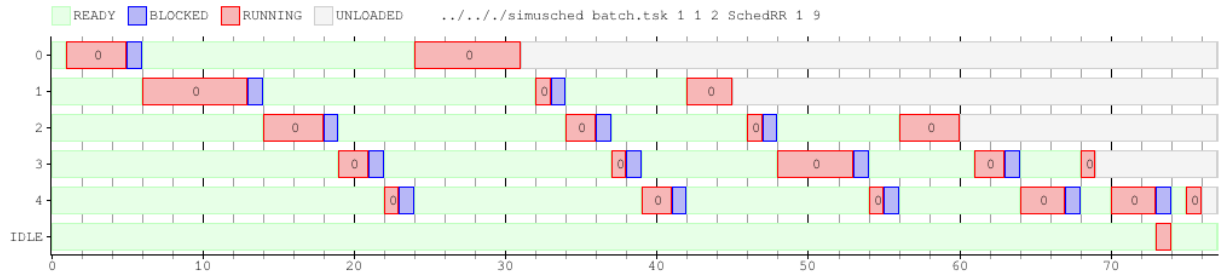

Figura 9: Round Robin con quantum 9

Turnarround: 56.2 ciclos

Waiting time: 42.2 ciclos

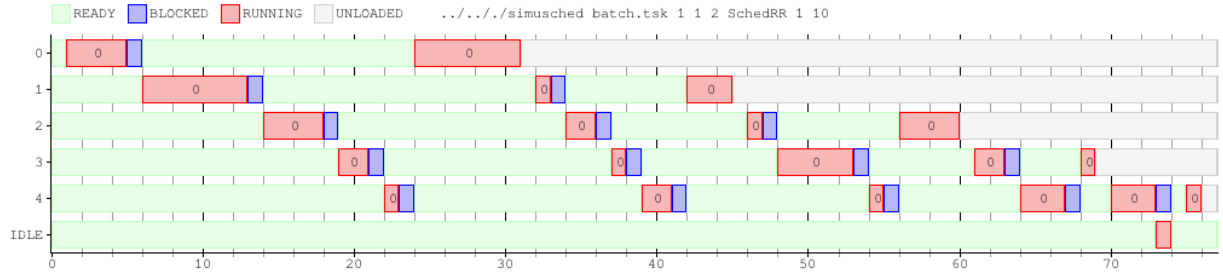

Figura 10: Round Robin con quantum 10

Turnarround: 56.2 ciclos

Waiting time: 42.2 ciclos

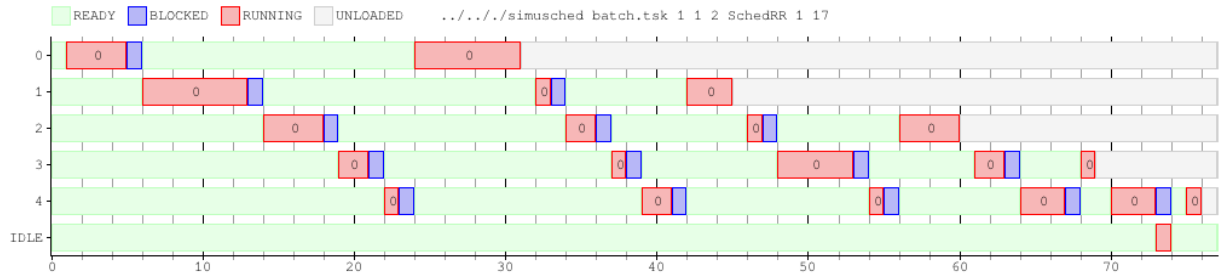

Figura 11: Round Robin con quantum 17

Turnarround: 56.2 ciclos

Waiting time: 42.2 ciclos

Como se puede observar ambas métricas mejoran mientras en quantum es mas alto, pero luego de tener quantum 9, las métricas dan lo mismo (por dar el mismo gráfico). Por lo tanto el valor óptimo de quantum es 9 para este tipo de test.

7.3. Experimento con 2 cores

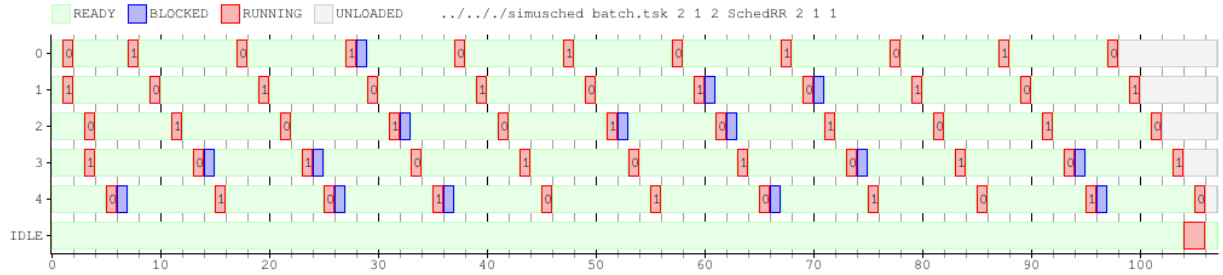

Figura 12: Round Robin con quantum 1

Turnarround: 102 ciclos

Waiting time: 88 ciclos

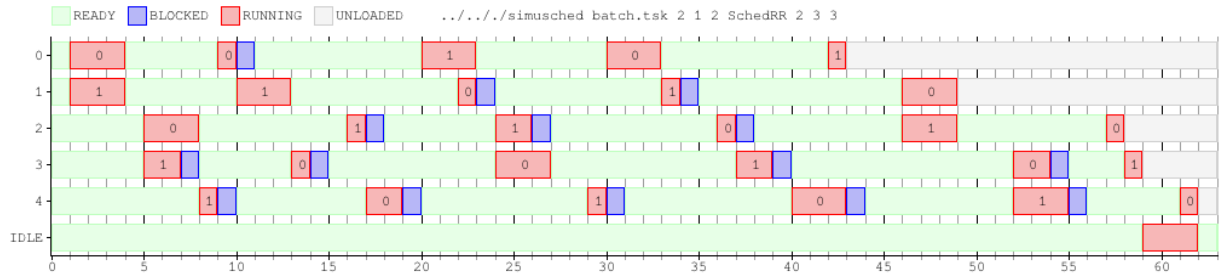

Figura 13: Round Robin con quantum 3

Turnarround: 54.4 ciclos

Waiting time: 40.2 ciclos

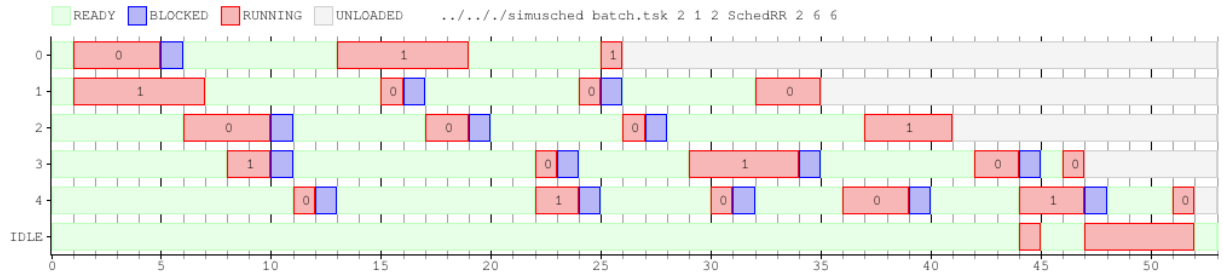

Figura 14: Round Robin con quantum 6

Turnarround: 40.2 ciclos

Waiting time: 26.2 ciclos

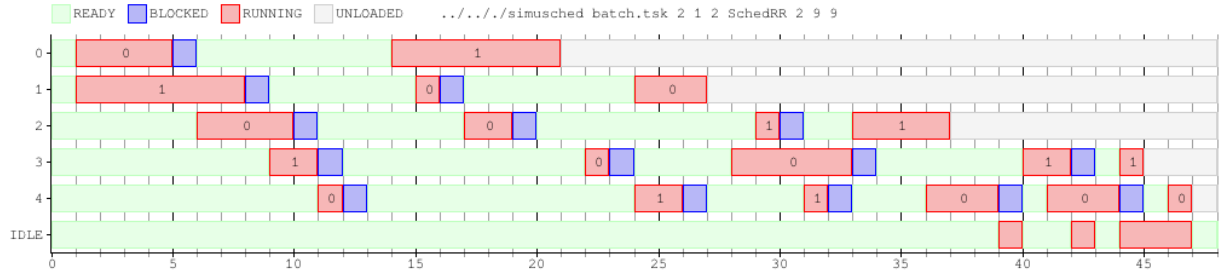

Figura 15: Round Robin con quantum 9

Turnarround: 35.4 ciclos

Waiting time: 21.4 ciclos

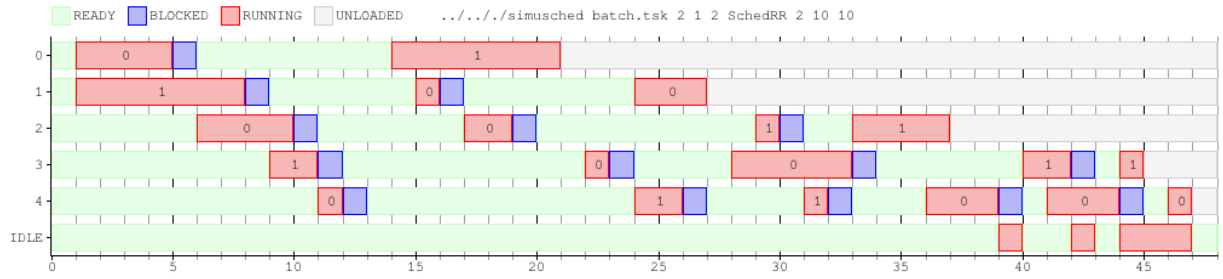

Figura 16: Round Robin con quantum 10

Turnarround: 35.4 ciclos

Waiting time: 21.4 ciclos

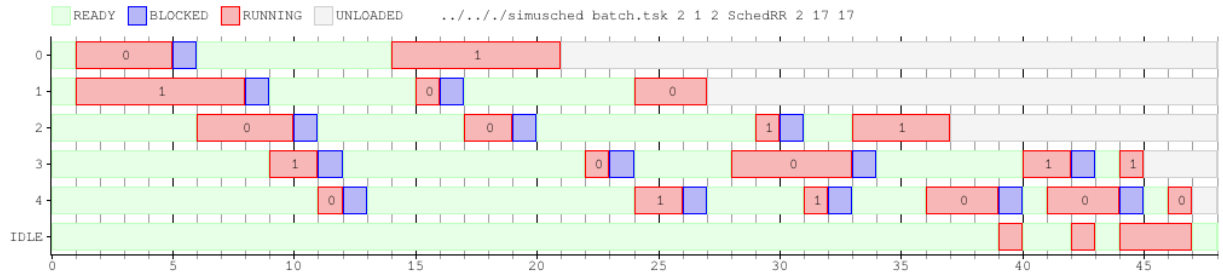

Figura 17: Round Robin con quantum 17

Turnarround: 56.2 ciclos

Waiting time: 42.2 ciclos

Como se puede observar ambas métricas mejoran mientras en quantum es mas alto, pero luego de tener quantum 9, las métricas dan lo mismo (por dar el mismo gráfico). Por lo tanto el valor óptimo de quantum es 9 para este tipo de test.

7.4. Experimento con 4 cores

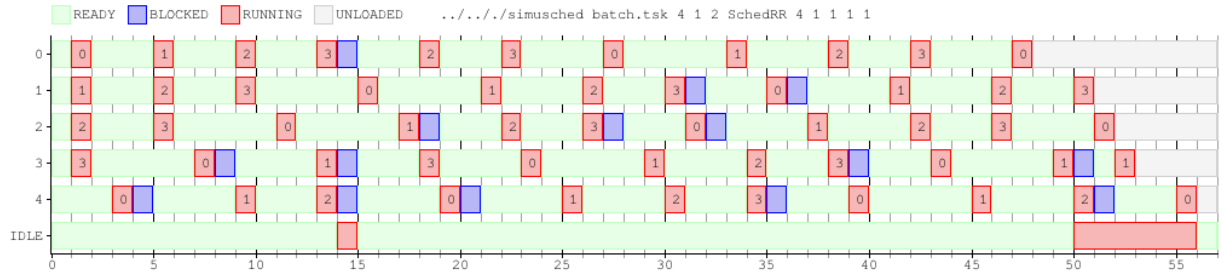

Figura 18: Round Robin con quantum 1

Turnarround: 51 ciclos

Waiting time: 38 ciclos

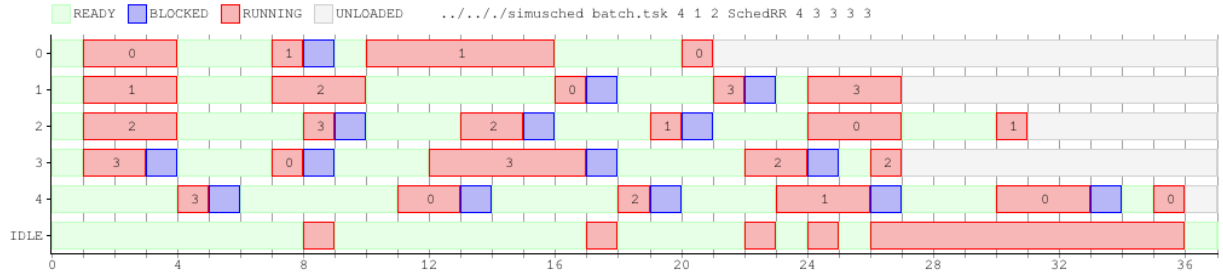

Figura 19: Round Robin con quantum 3

Turnarround: 28.4 ciclos

Waiting time: 14.4 ciclos

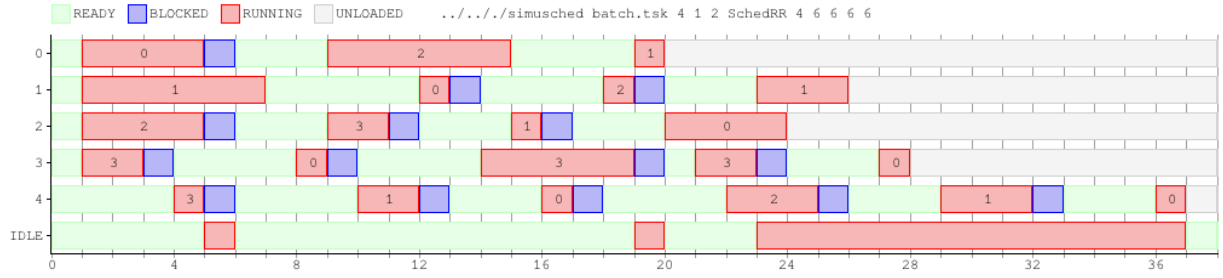

Figura 20: Round Robin con quantum 6

Turnarround: 27 ciclos

Waiting time: 13 ciclos

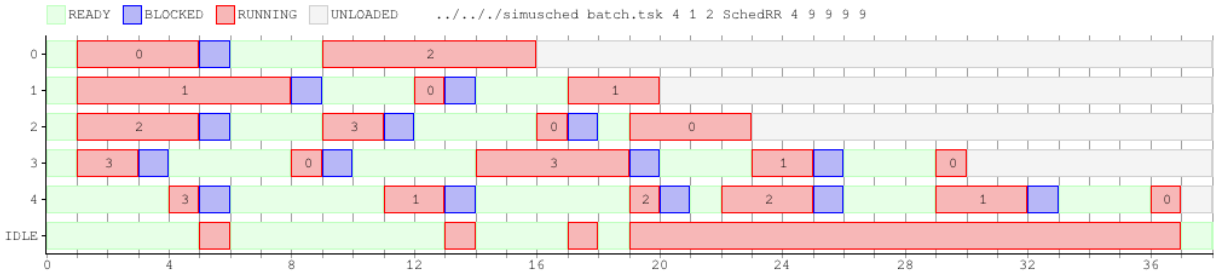

Figura 21: Round Robin con quantum 9

Turnarround: 25 ciclos

Waiting time: 10.2 ciclos

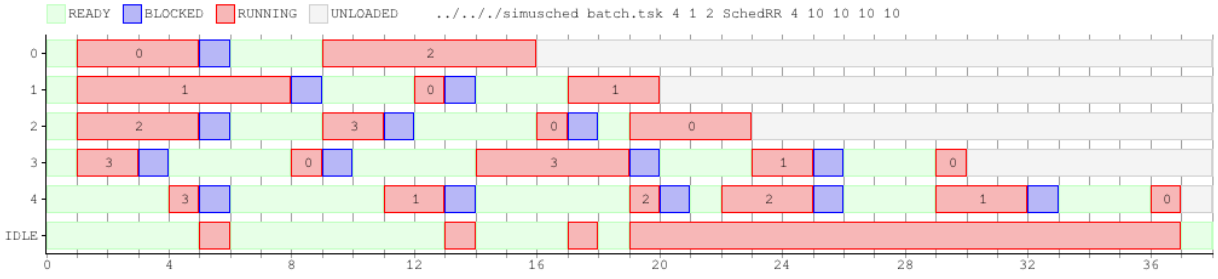

Figura 22: Round Robin con quantum 10

Turnarround: 25 ciclos

Waiting time: 10.2 ciclos

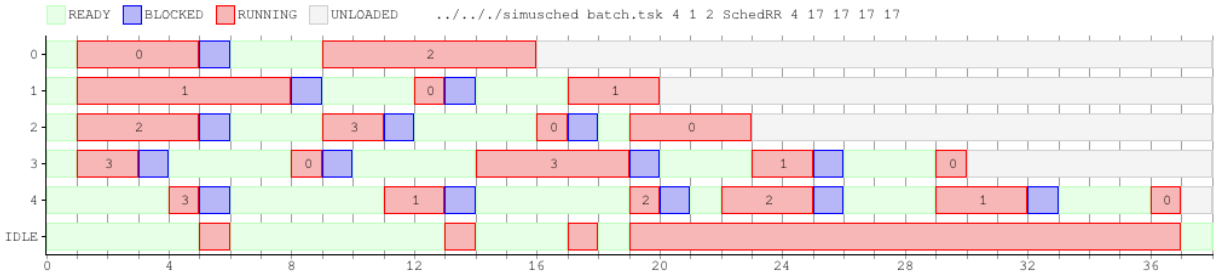

Figura 23: Round Robin con quantum 17

Turnarround: 56.2 ciclos

Waiting time: 42.2 ciclos

Como se puede observar ambas métricas mejoran mientras en quantum es mas alto, pero luego de tener quantum 9, las métricas dan lo mismo (por dar el mismo gráfico). Por lo tanto el valor óptimo de quantum es 9 para este tipo de test.

8. Pregunta 8

8.1. Enunciado del Problema:

Implemente un scheduler Round-Robin que no permita la migración de procesos entre núcleos (SchedRR2). La asignación de CPU se debe realizar en el momento en que se produce la carga de un proceso (load). El nucleo correspondiente a un nuevo proceso sera aquel con menor cantidad de procesos activos totales (RUNNING + BLOCKED + READY). Disene y realice un conjunto de experimentos que permita evaluar comparativamente las dos implementaciones de Round-Robin.

8.2. Explicación de la implementacion

Para implementar esta política de scheduling utilizamos las siguientes estructuras:

- Un vector para almacenar el valor del quantum de cada núcleo del procesador.
- Un vector que contiene la cola de procesos *ready* de cada núcleo.
- Un vector que almacena cuantos procesos bloqueados hay en cada núcleo.
- Un diccionario que indica a que núcleo esta asignado cada proceso.

Cada vez que un proceso se añade al scheduler calculamos, para cada núcleo, la cantidad de procesos activos, es decir: $RUNNING + BLOCKED + READY$. El núcleo con tenga el menor valor en dicha suma sera el núcleo asignado al nuevo proceso. Entonces, en el diccionario añadimos una entrada para la clave del *pid* del proceso y como valor colocamos el núcleo elegido. De esta forma siempre sabemos a que núcleo fue asignado el proceso.

Cuando un proceso se bloquea, incrementamos en uno el contador de procesos bloqueados del núcleo en el que el proceso se ejecuto. Cuando se desbloquea, lo disminuimos en uno y colocamos el proceso en la cola de procesos *ready* del núcleo que se le asigno al ser cargado.

Cuando debemos determinar el próximo proceso a ejecutar en un núcleo, simplemente tomamos el primer elemento de la cola de procesos *ready* correspondiente a dicho núcleo.

8.3. Comparación entre los algoritmos de Round Robin

En todas las pruebas subsecuentes utilizamos la siguiente configuración del simulador:

- **Dos (2) núcleos**. Ya que con un solo núcleo los algoritmos se comportarían exactamente igual y con dos núcleos ya es posible mostrar las diferencias en el comportamiento.
- **Costo de cambio de contexto**: Un (1) ciclo de reloj.
- **Costo de migración de núcleo**: Dos (2) ciclos de reloj.
- **Quantum de cada núcleo**: Dos (2) ciclos de reloj.

8.3.1. Primer experimento

En este primer experimento solo ejecutamos tareas que utilizan intensivamente el CPU durante diferente tiempo. El lote utilizado es el siguiente:

@1
TaskCPU 15
@3
TaskCPU 5
TaskCPU 15
@4
TaskCPU 2

El resultado de utilizar RR y RR2 fue:

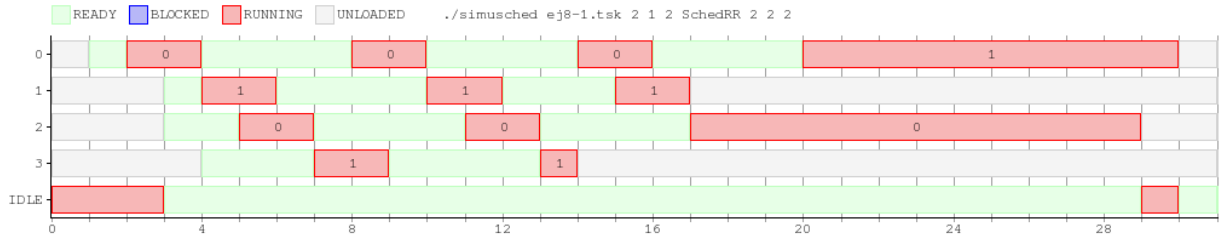

Figura 24: Round Robin con migración entre núcleos

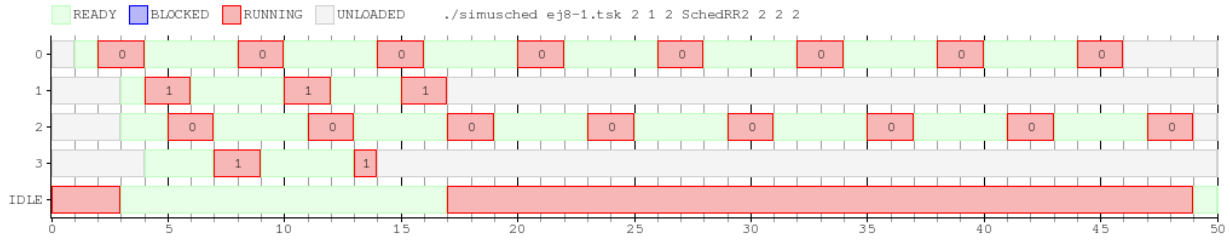

Figura 25: Round Robin sin migración entre núcleos

Como se puede observar en los gráficos, la migración entre núcleos permite al primer scheduling terminar mucho mas rápido *(20 ciclos antes, aunque esta diferencia podría ser arbitrariamente grande)* que el segundo, que no permite migrar procesos entre núcleos. Esto se debe a que, en el segundo scheduler, se asigna el núcleo cero a las dos tareas mas largas *(tareas 0 y 2)*. Entonces, al no estar permitida la migración entre núcleos, el núcleo cero debe ejecutar ambas tareas mientras el núcleo uno esta ocioso *(lo cual es algo totalmente indeseado)*.

8.3.2. Segundo experimento

En el segundo experimento utilizamos tareas que utilizan intensivamente la E/S. El lote utilizado fue el siguiente:

@1
TaskConsola 6 3 3
@3
TaskConsola 5 5 5
TaskConsola 4 7 7
@4
TaskConsola 6 4 4

El resultado de utilizar RR y RR2 fue:

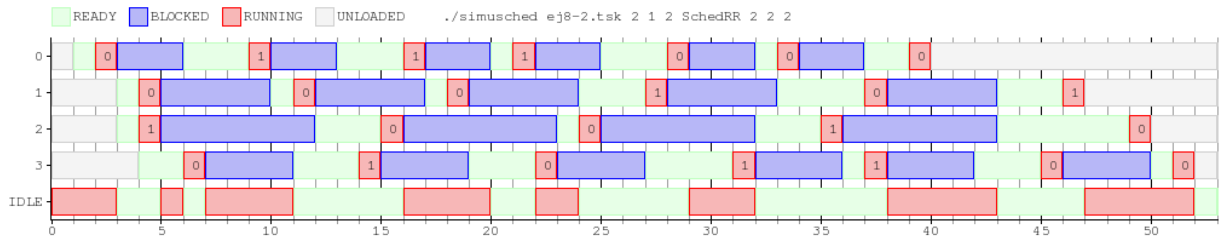

Figura 26: Round Robin con migración entre núcleos

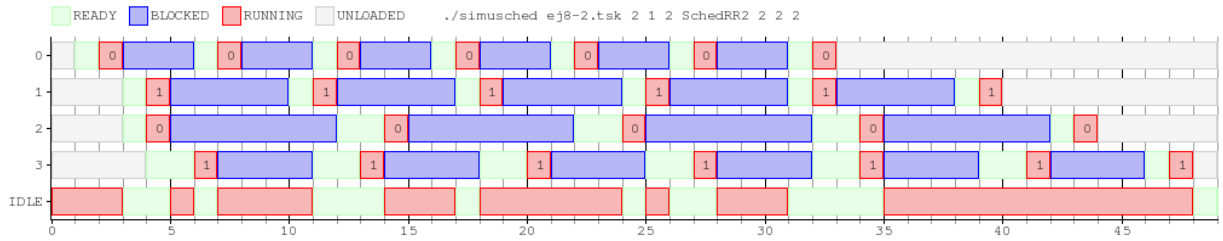

Figura 27: Round Robin sin migración entre núcleos

Como se puede ver en los gráficos, en este caso el scheduler Round Robin que no permite la migración entre núcleos termina antes que el si lo permite. Esto es debido a que el costo de migrar un proceso de un núcleo a otro es mayor que el costo que tendría esperar a que el núcleo que se le asigno previamente quede libre, pues las tareas se bloquean muy rápidamente.

8.3.3. Conclusión

En algunos casos, puede resultar beneficioso que un proceso espere al núcleo que tenia asignado previamente antes que realizar un migración. Pero, si dicho núcleo no se libera rápidamente y, ademas, teníamos otro núcleo libre, hubiera sido mejor pagar el costo de una migración entre núcleos.

En el primer experimento solo teníamos una tarea a la espera de que su núcleo asignado se libere, mientras había otro núcleo libre. Pero, fácilmente, podría generarse un escenario donde la cantidad de procesos a la espera de un núcleo sea arbitrariamente grande y que la cantidad

de núcleos libres también sea arbitrariamente grande, dando como resultado una asignación muy ineficiente de los recursos disponibles.

Por lo dicho, podemos concluir que permitir la migración entre núcleos es, en la mayoría de los casos, mas conveniente que prohibirlo.

9. Pregunta 9

9.1. Enunciado del Problema:

Disenar y llevar a cabo un experimento que permita poner a prueba la ecuanimidad (fairness) del algoritmo SchedLottery implementado. Tener en cuenta que, debido al factor pseudoaleatorio involucrado, cualquier corrida puntual podria ser arbitrariamente injusta; sin embargo, si se repite un mismo experimento n veces y se observan los resultados acumulativos, tales anomalias deberian ir desapareciendo conforme n aumenta. En otras palabras, interesa mostrar en base a evidencia empirica que el algoritmo implementado efetivamente tiende a ser totalmente ecuanime a medida que n tiende a infinito.

9.1.1. Introducción

Para poner a prueba la ecuanimidad (fairness) del *Lottery Scheduling* fijaremos los siguiente parámetros:

- Quantum de 4 ciclos de clock. Este valor no afecta la ecuanimidad.
- Costo de cambio de contexto y de migración entre núcleos: 0 ciclos de clock, para que no entorpezca las mediciones y, ademas, es un valor que no afecta el *fairness*.

La experimentación la realizamos con uno y dos núcleos de procesamiento, para ver si el *fairness* se da en ambos casos.

Vamos a utilizar tanto tareas que utilizan intensivamente el CPU como tareas que utilizan la E/S. El lote de tareas elegido es el siguiente:

TaskCPU 25
TaskCPU 22
TaskBatch 14 7
TaskBatch 12 9
TaskConsola 10 2 2

Elegimos este lote para tener variedad tanto en el tiempo de ejecución de los procesos como en el uso de la CPU y de la E/S que estos realizan.

La métrica elegida para medir el *fairness* es *promedio del uso del CPU por tiempo de espera de cada proceso*, es decir, llevamos a cabo la siguiente cuenta para determinar el valor de la métrica en un experimento:

- Para cada proceso i calculamos cuantos ciclos del CPU utilizo el proceso, lo llamamos CPU_i y cuanto tiempo espero, lo llamamos T_i.
- Luego, el valor de la métrica para el experimento es: $\frac{1}{n} * \sum_{i=1}^{n} \frac{CPU_i}{T_i}$

El concepto de *fairness* en el lottery scheduling es:

- Cuanto menos usa de su quantum un proceso antes de bloquearse, mas probable es que el proceso gane el CPU *pronto* (ya que se le otorga un *compensation ticket*), es decir, que a menor uso del quantum menor debería ser el tiempo de espera antes de volver a tener el CPU.
- Si un proceso utiliza todo su quantum, es probable que deba esperar mas tiempo que los procesos que solo utilizan una fracción de este, es decir, a mayor uso del CPU mayor debería ser el tiempo de espera antes de volver a tener el CPU.

Luego, si **en promedio** todos los procesos tienen un relación similar entre el uso del CPU y el tiempo que esperan (a lo largo de los experimentos) podríamos decir que el *lottery scheduling* es justo asignando el CPU. Justamente, la métrica elegida nos permite ver esta relación.

Es importante señalar que la métrica utilizada no se ve afectada por el tiempo que el proceso esta bloqueado, ya que los dos valores que determinan su valor (uso del CPU y tiempo de espera) son independientes del tiempo que el proceso esta bloqueado.

En cada conjunto de experimentos, decidimos llevar a cabo la medición de la métrica hasta un determinado tick del procesador que nos garantice que aun no haya terminado de ejecutarse ningún proceso (es decir, que todos compiten por el CPU) y, ademas, independiza la experimentación de los tiempos de ejecución totales de los procesos en el lote de procesos.

9.1.2. Experimentación con un núcleo

A continuación los resultados de la experimentación realizada utilizando diferentes semillas para el generador de números aleatorios.

Para cada experimento, calculamos la métrica para cada proceso, es decir, el valor $\frac{CPU_i}{T_i}$ y la métrica para el experimento, es decir, el promedio de los valores anteriores.

En este experimento decidimos llevar a cabo la medición hasta el tick numero 40 del procesador que, como dijimos antes, nos garantiza que ningún procesos haya terminado. A continuación, los resultados de la experimentación realizada:

TICK 40													
		Tiempo de espera						Ticks de CPU					Metrica promedio
Semilla	# Nucleos	P1	P2	P3	P4	P5		P1	P2	P3	P4	P5	
1	1	28	36	30	19	24		12	4	6	12	6	0,3242522974
2	1	28	32	24	27	31		12	8	10	7	3	0,2902543096
3	1	32	24	25	31	37		8	16	10	5	1	0,3009968033
4	1	28	32	34	22	22		12	8	4	10	6	0,3046982429
5	1	20	38	34	27	19		20	2	4	7	7	0,3595917899
6	1	30	36	23	27	19		10	4	11	8	7	0,3174845326
7	1	24	28	36	31	30		16	12	3	5	4	0,2946390169
8	1	29	28	21	40	28		11	12	13	0	4	0,3139573071
9	1	28	32	24	30	31		12	8	11	6	3	0,2867357911
10	1	36	20	30	32	28		4	20	7	5	4	0,3287103175
11	1	36	24	29	25	28		4	16	8	8	4	0,3032993979
12	1	32	20	29	34	38		8	20	8	3	1	0,3280826305
13	1	28	24	34	36	21		12	16	3	2	7	0,3144724556
14	1	36	20	28	30	31		4	20	7	6	3	0,3315770609
15	1	32	24	27	30	37		8	16	9	6	1	0,2954054054
16	1	35	28	31	25	28		5	12	7	8	4	0,2520184332
17	1	28	31	30	28	25		12	9	7	7	5	0,2804454685
18	1	32	28	27	28	28		8	12	9	7	4	0,280952381
19	1	28	32	30	23	32		12	8	7	10	3	0,2880874741
20	1	32	20	33	33	31		8	20	5	4	3	0,3239002933
21	1	31	28	32	25	25		9	12	5	9	5	0,2870288018
22	1	36	18	30	38	22		4	22	7	1	6	0,3731419458
23	1	36	36	22	22	14		4	4	12	11	9	0,3821067821
24	1	24	38	19	27	37		16	2	14	7	1	0,3484853274
25	1	31	24	34	32	23		9	16	4	5	6	0,2983511743
26	1	20	33	33	32	28		20	7	4	5	4	0,3264880952
27	1	32	28	25	32	28		8	12	11	5	4	0,2835357143
28	1	20	32	32	33	34		20	8	6	4	2	0,3235071301
29	1	20	28	35	36	33		20	12	3	2	3	0,3321500722
30	1	24	36	27	26	28		16	4	8	8	4	0,3049247049

Figura 28: Lottery Scheduling con un solo núcleo hasta el tick 40

Como puede apreciarse en la tabla, la métrica da valores muy similares para todos los experimentos realizados. Para ver que relación existe entre estos valores calculamos la varianza, la esperanza y el desvió estándar de la muestra, lo cuales se definen como:

$$\text{Esperanza} = \mu = \bar{X} = \frac{1}{n} * \sum_{i=1}^{n}(X_i)$$
$$\text{Varianza} = \sigma^2 = \frac{1}{n} * \sum_{i=1}^{n}(X_i - \bar{X})^2$$
$$\text{Desvió estandar} = \sigma = \sqrt{Varianza}$$

Esperanza	0,3126427052
Varianza	0,00082674855
Desvio estandar	0,02875323564

Figura 29: Esperanza, varianza y desvió estándar de la métrica promedio

Como se puede ver, la varianza y el desvió estándar son muy chicos en comparación con la esperanza. Numéricamente hablando, la varianza es un 0,26 % de la esperanza y el desvió estándar es un 9,20 % de la esperanza.

Entonces, como el valor esperado (i.e esperanza) de la métrica se encuentra dentro de un intervalo muy pequeño $(\mu-\sigma, \mu+\sigma) = (0{,}28, 0{,}34)$, podemos concluir, en base a la experimentación realizada y a la métrica utilizada, que este mismo comportamiento ocurrirá en la mayoría de los experimentos y que, por lo tanto, el *Lottery scheduling* es justo.

9.1.3. Experimentación con dos núcleos

En este experimento decidimos llevar a cabo la medición hasta el tick numero 25 del procesador, por los motivos ya dichos anteriormente. A continuación, los resultados de la experimentación realizada:

TICK 40													
		Tiempo de espera						Ticks de CPU					Metrica promedio
Semilla	# Nucleos	P1	P2	P3	P4	P5		P1	P2	P3	P4	P5	
1	1	28	36	30	19	24		12	4	6	12	6	0,3242522974
2	1	28	32	24	27	31		12	8	10	7	3	0,2902543096
3	1	32	24	25	31	37		8	16	10	5	1	0,3009968033
4	1	28	32	34	22	22		12	8	4	10	6	0,3046982429
5	1	20	38	34	27	19		20	2	4	7	7	0,3595917899
6	1	30	36	23	27	19		10	4	11	8	7	0,3174845326
7	1	24	28	36	31	30		16	12	3	5	4	0,2946390169
8	1	29	28	21	40	28		11	12	13	0	4	0,3139573071
9	1	28	32	24	30	31		12	8	11	6	3	0,2867357911
10	1	36	20	30	32	28		4	20	7	5	4	0,3287103175
11	1	36	24	29	25	28		4	16	8	8	4	0,3032993979
12	1	32	20	29	34	38		8	20	8	3	1	0,3280826305
13	1	28	24	34	36	21		12	16	3	2	7	0,3144724556
14	1	36	20	28	30	31		4	20	7	6	3	0,3315770609
15	1	32	24	27	30	37		8	16	9	6	1	0,2954054054
16	1	35	28	31	25	28		5	12	7	8	4	0,2520184332
17	1	28	31	30	28	25		12	9	7	7	5	0,2804454685
18	1	32	28	27	28	28		8	12	9	7	4	0,280952381
19	1	28	32	30	23	32		12	8	7	10	3	0,2880874741
20	1	32	20	33	33	31		8	20	5	4	3	0,3239002933
21	1	31	28	32	25	25		9	12	5	9	5	0,2870288018
22	1	36	18	30	38	22		4	22	7	1	6	0,3731419458
23	1	36	36	22	22	14		4	4	12	11	9	0,3821067821
24	1	24	38	19	27	37		16	2	14	7	1	0,3484853274
25	1	31	24	34	32	23		9	16	4	5	6	0,2983511743
26	1	20	33	33	32	28		20	7	4	5	4	0,3264880952
27	1	32	28	25	32	28		8	12	11	5	4	0,2835357143
28	1	20	32	32	33	34		20	8	6	4	2	0,3235071301

Figura 30: Lottery Scheduling con un dos núcleos hasta el tick 25

Al igual que sucedió en la experimentación de un solo núcleo, los valores obtenidos con la métrica son muy similares. Para ver que relación hay entre estos vamos a calcular nuevamente, la esperanza, la varianza y el desvió estándar de la muestra. Los resultados obtenidos son los siguientes:

Esperanza	0,3126427052
Varianza	0,00082674855
Desvio estandar	0,02875323564

Figura 31: Esperanza, varianza y desvió estándar de la métrica promedio

Como se puede ver, la varianza es un 2,75 % de la esperanza y el desvió estándar es un 15,75 % de la esperanza y, el desvió estándar, nos dice que la esperanza se encuentra en el intervalo $(\mu - \sigma, \mu + \sigma) = (0{,}93,\ 1{,}28)$. Por esto y por el hecho de que el intervalo en el que se encuentra la esperanza es pequeño, podemos suponer que el resultado obtenido en la experimentación realizada se repetirá en la mayoría de los casos y que, en consecuencia, el *Lottery scheduling* es justo.

9.1.4. Conclusión

En general, el Lottery Scheduling se comporta de forma ecuánime, como podemos ver en los experimentos, ya que los intervalos de la esperanza son muy pequeños con ambos núcleos y la varianza también es pequeña. Por otro lado por el factor aleatorio de este algoritmo, es posible que se den casos extremos en donde una tarea o proceso es pospuesto por mucho tiempo o que una tarea gane el uso del CPU muy seguido.

10. Pregunta 10

10.1. Enunciado del Problema:

Los autores del articulo sobre lottery scheduling alegan que la optimizacion de compensation tickets es necesaria para compensar una posible falencia del algoritmo inicialmente propuesto en ciertos escenarios. Disenar y llevar a cabo un experimento apropiado para comprobar esta afirmacion (provocar un escenario donde se manifieste el problema, comparar simulaciones ejecutadas con y sin compensation tickets y discutir los resultados obtenidos).

10.2. Solución

Dado que lo que queremos ver es que la optimizacion de *Compensation tickets* es necesaria, mostraremos que en un escenario propuesto por nosotros el *lottery scheduling* es menos justo con la asignación del CPU a los procesos si no utiliza el *compensation ticket*.

El escenario propuesto es el siguiente:

- Fijaremos el quantum en 4 ciclos de clock.
- Fijaremos el costo de cambio de contexto y el costo de cambio de núcleo en cero ciclos de clock para que no entorpezcan las mediciones.
- Fijaremos el numero de núcleos de procesamiento en uno.

Ademas, utilizaremos el siguiente lote de tareas:

```
TaskCPU 35
TaskCPU 35
TaskBatch 9 9
```

Elegimos este lote para tener dos procesos que utilizan intensivamente el CPU compitiendo continuamente por el CPU y que, al deshabilitar el *Compensation Ticket*, obliguen al proceso TaskBatch a esperar mucho antes de poder utilizar el CPU.

Ademas, cada vez que el proceso TaskBatch gane el CPU, lo utilizara por un solo ciclo y luego se bloqueara (por como esta implementado TaskBatch). Se hace eligio esto para minimizar la cantidad de quantum que la tareas bloqueante utiliza y así maximizar la cantidad de tickets que obtendrá el proceso por los Compensation Tickets.

Es importante señalar que probamos con muchos otros lotes de tareas y, en todos ellos, obteníamos el resultado inverso al esperado, es decir, la experimentación daba como resultado que el algoritmo sin compensation ticket era mas justo que con. Con este lote de tareas no sucede eso, y ese fue otro de los motivos por el cual lo elegimos.

Utilizaremos la métrica descrita en el **Ejercicio 9** para medir el *fairness* del algoritmo, y llevaremos a cabo la medición hasta el tick 42 del procesador, por los motivos que explicamos en el ejercicio anterior.

A continuación, exponemos los resultados de la experimentación utilizando el Compensation Ticket y el lote descrito:

CON Compensation Ticket									
		T de Esper				Ticks CPU			Metrica
Semilla	# Nucleos	P1	P2	P3		P1	P2	P3	
1	1	25	18	40		17	24	1	0,6794444444
2	1	14	31	36		28	11	3	0,8127240143
3	1	32	18	30		10	24	6	0,6152777778
4	1	34	10	39		8	32	2	1,162192056
5	1	26	19	36		16	23	3	0,6364147548
6	1	31	13	36		11	29	3	0,8896470913
7	1	17	26	40		25	16	1	0,7036576169
8	1	32	16	30		10	26	6	0,7125
9	1	13	21	32		29	21	5	1,12900641
10	1	16	30	34		26	12	4	0,7142156863
11	1	25	20	32		17	22	5	0,6454166667
12	1	24	22	34		18	20	4	0,5922459893
13	1	30	18	15		12	24	27	1,177777778
14	1	34	11	36		8	31	3	1,04560309
15	1	14	30	38		28	12	2	0,8175438596
16	1	30	25	32		12	17	5	0,4120833333
17	1	26	21	32		16	21	5	0,5905448718
18	1	17	30	32		25	12	5	0,6756127451
19	1	18	28	34		24	14	4	0,6503267974
20	1	29	14	38		13	28	2	0,8336358137
21	1	21	26	32		21	16	5	0,5905448718
22	1	31	14	36		11	28	3	0,8127240143
23	1	22	24	34		20	18	4	0,5922459893
24	1	30	15	36		12	27	3	0,7611111111
25	1	22	22	39		20	20	2	0,6231546232
26	1	18	26	38		24	16	2	0,6671165092
27	1	18	26	39		24	16	2	0,6666666667
28	1	20	26	34		22	16	4	0,6110105581
29	1	18	27	36		24	15	3	0,6574074074
30	1	22	26	31		20	16	6	0,5726746372

Figura 32: Lottery Scheduling con Compensation Tickets hasta el tick 42

Ahora, si llevamos a cabo la misma experimentación, pero **sin Compensation Ticket** resulta en el siguiente gráfico:

		SIN Compensation Ticket							
		T de espera				Ticks CPU			Metrica
Semilla	# Nucleos	P1	P2	P3		P1	P2	P3	
1	1	26	31	34		16	11	4	0,3626234613
2	1	26	22	31		16	20	6	0,5726746372
3	1	31	18	28		11	24	7	0,6460573477
4	1	34	10	38		8	32	2	1,162641899
5	1	30	18	30		12	24	6	0,6444444444
6	1	37	14	32		5	28	5	0,763795045
7	1	34	27	32		8	15	5	0,3156998911
8	1	34	15	28		8	27	7	0,7617647059
9	1	26	21	32		16	21	5	0,5905448718
10	1	20	26	34		22	16	4	0,6110105581
11	1	30	21	24		12	21	9	0,5916666667
12	1	24	22	34		18	20	4	0,5922459893
13	1	34	15	28		8	27	7	0,7617647059
14	1	22	26	30		20	16	6	0,5748251748
15	1	6	38	38		36	4	2	2,052631579
16	1	27	22	28		15	20	7	0,5715488215
17	1	15	34	28		27	8	7	0,7617647059
18	1	19	26	36		23	16	3	0,6364147548
19	1	26	24	26		16	18	8	0,5576923077
20	1	26	17	40		16	25	1	0,7036576169
21	1	18	28	34		24	14	4	0,6503267974
22	1	26	21	32		16	21	5	0,5905448718
23	1	26	23	28		16	19	7	0,5638238573
24	1	32	18	26		10	24	8	0,6511752137
25	1	22	22	38		20	20	2	0,6236044657
26	1	22	26	30		20	16	6	0,5748251748
27	1	22	22	38		20	20	2	0,6236044657
28	1	34	17	24		8	25	9	0,693627451
29	1	24	26	26		18	16	8	0,5576923077
30	1	28	22	26		14	20	8	0,5722610723

Figura 33: Scheduling lottery sin Compensation Tickets hasta el tick 42

Ahora, para comparar los resultados obtenidos en ambos experimentos, calcularemos la esperanza, la varianza y el desvió estándar de ambas muestras. El resultado es el siguiente:

SIN ticket:	
Esperanza	0,6382766751
Varianza	0,07919273876
Desvio estandar	0,2814120445
CON ticket:	
Esperanza	0,8055079852
Varianza	0,03855961702
Desvio estandar	0,1963660282

Figura 34: Esperanza, Varianza, y Desvió estándar de las muestras

Como se puede observar, tanto la varianza como el desvió estándar son mayores cuando el algoritmo no utiliza el Compensation Ticket, es decir, el intervalo en el cual se haya la esperanza $(\mu-\sigma, \mu+\sigma)$ es mayor y, por lo tanto, podemos concluir que el algoritmo es menos justo asignando el CPU. Luego, podemos concluir que en ciertos escenarios, la optimizacion del Compensation Ticket es necesaria, pues sin el Compensation ticket la asignación del CPU es menos justa.