DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº 1

Primer cuatrimestre de 2016 — Sistemas Operativos

Integrante	LU	Correo electrónico
Martínez, Manuela	160/14	martinez.manuela.22@gmail.com
Rabinowicz, Lucía	105/14	lu.rabinowicz@gmail.com
Goldstein, Brian	027/14	brai.goldstein@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Ejercicio 1

TaskConsola(int pid, vector(int) params)

```
1 int r
2 para (i = 1; i ≤ params[0]; i++) hacer
3 |   r ← random(params[1], params[2])
4 |   uso_IO(pid, r)
5 fin
```

Para implementar esta tarea interactiva se utilizó un ciclo donde en cada iteración se genera un nuevo número aleatorio entre los valores pasados por parámetro (utilizando la función rand) que representa la duración de la llamada bloqueante. Este proceso se realiza n veces.

A continuación se presentan el lote utilizado y su correspondiente gráfico.

```
TaskConsola 5 2 5
@3:
TaskConsola 2 1 6
@4:
*2 TaskConsola 3 1 3
```

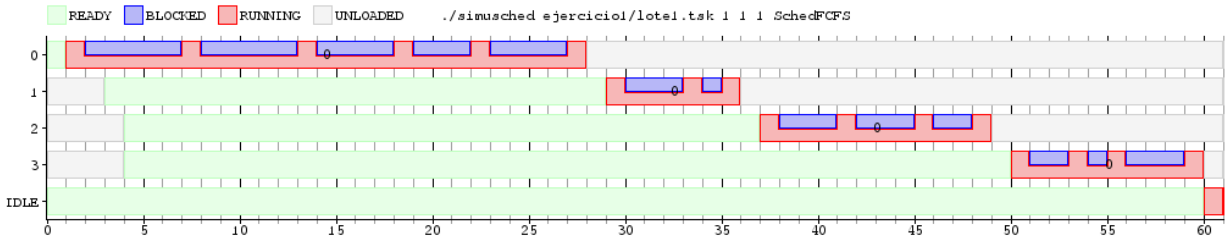

Figura 1: Scheduler: FCFS

Para ver que el random se realiza de manera correcta su ejecuto 100 veces una misma tarea que se bloquea una única vez y veremos que el tiempo de bloqueo es realmente aleatorio. La cantidad de veces que se puso a correr la tarea fue elegida de forma aleatoria.
A continuación se presentan el lote utilizado y el histograma:

```
TaskConsola 1 1 30
```

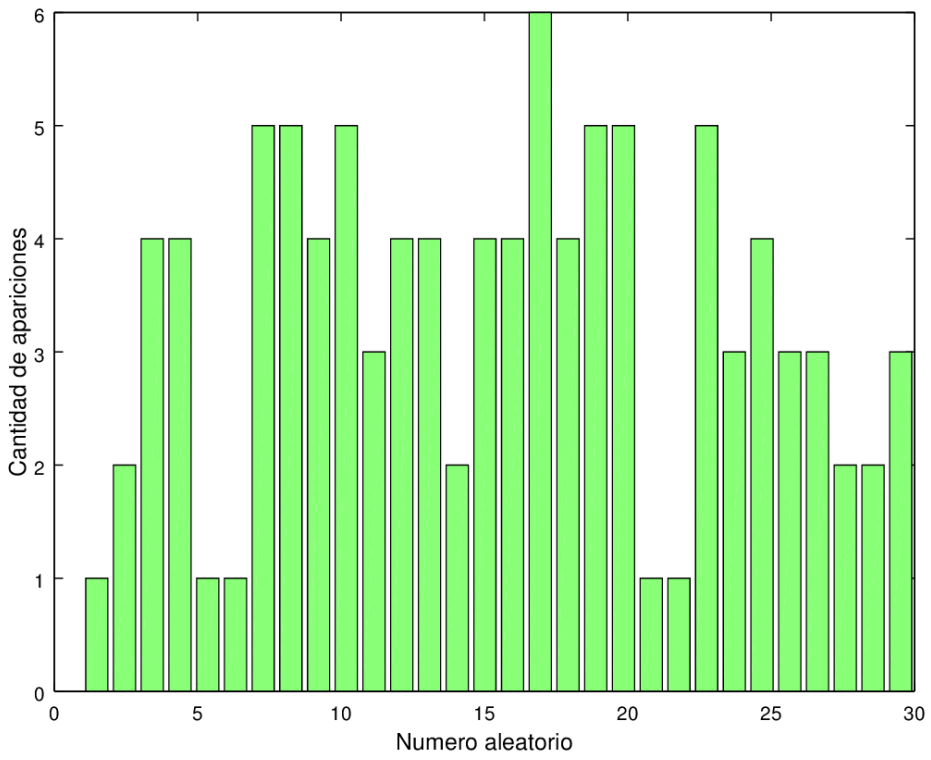

Figura 2: Histograma

2. Ejercicio 2

TaskEj2(int pid, vector(int) params)

```
1 uso_cpu(pid, params[0] - 1)
2 int r
3 r ← random(1, 4)
4 uso_IO(pid, r)
5 return
```

Para este ejercicio se creó una nueva tarea llamada TaskEj2 que recibe por parámetro el *pid* y además un entero n que representa el tiempo donde comenzará a ejecutarse la llamada bloqueante. Este valor será utilizado como parámetro de la función uso_CPU. A su vez el parámetro que recibe la función uso_I/O será un número aleatorio entre 1 y 4 que representará el tiempo que durará la llamada bloqueante.

A continuación se presentan el lote utilizado y sus correspondientes gráficos.

```
TaskCPU 500
TaskEj2 10
TaskEj2 20
TaskEj2 30
```

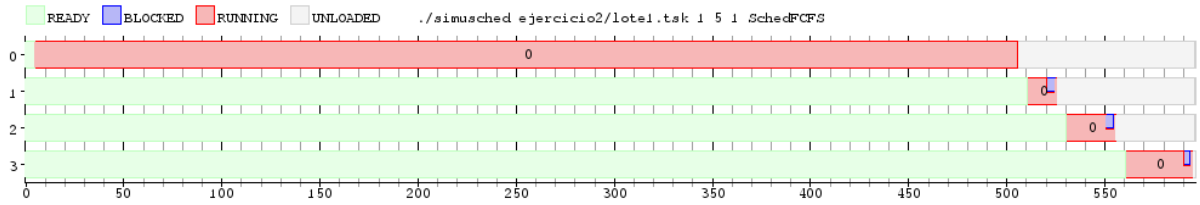

Figura 3: 1 núcleo

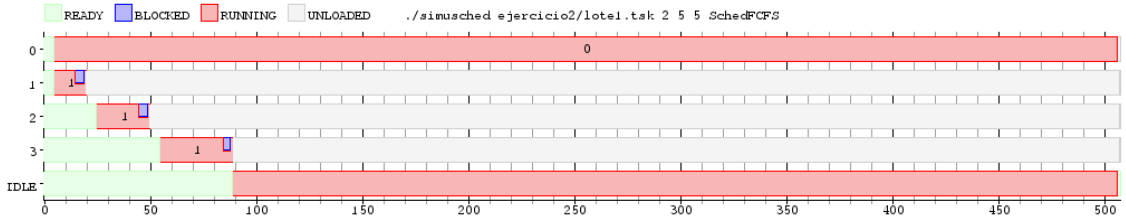

Figura 4: 2 núcleos

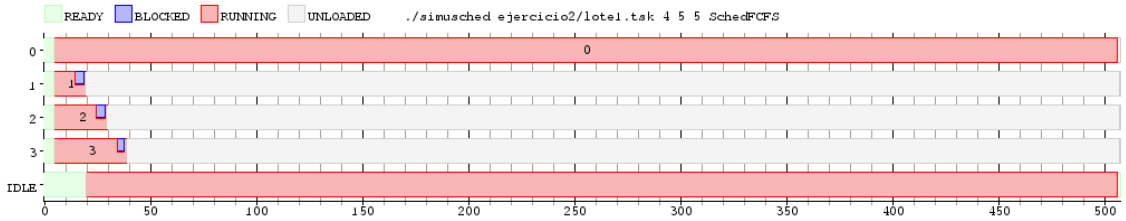

Figura 5: 4 núcleos

Cuadro 1: Latencia

Tarea	**1 núcleo**	**2 núcleos**	**4 núcleos**
0	0	0	0
1	506	0	0
2	526	20	0
3	556	50	0
Promedio	397	17,5	0

La desventaja que tendría el uso de este algoritmo de scheduling si solo se cuenta con una computadora con un único núcleo, como se puede ver en los gráficos, es que el tiempo que tomará en ejecutar las cuatro tareas será mayor que si se tuvieran más núcleos. Como se puede observar en el cuadro 1 a medida que se aumenta la cantidad de procesadores disminuye la latencia promedio. Esto se debe a que cuando se tienen mas núcleos las tareas se pueden ejecutar en paralelo y no es necesario esperar a que otra termine.

Para ver que el random se realiza de manera correcta su ejecuto 100 veces una misma tarea que se bloquea una única vez y veremos que el tiempo de bloqueo es realmente aleatorio. La cantidad de veces que se puso a correr la tarea fue elegida de forma aleatoria.
A continuación se presentan el lote utilizado y el histograma:

```
TaskEj2 10
```

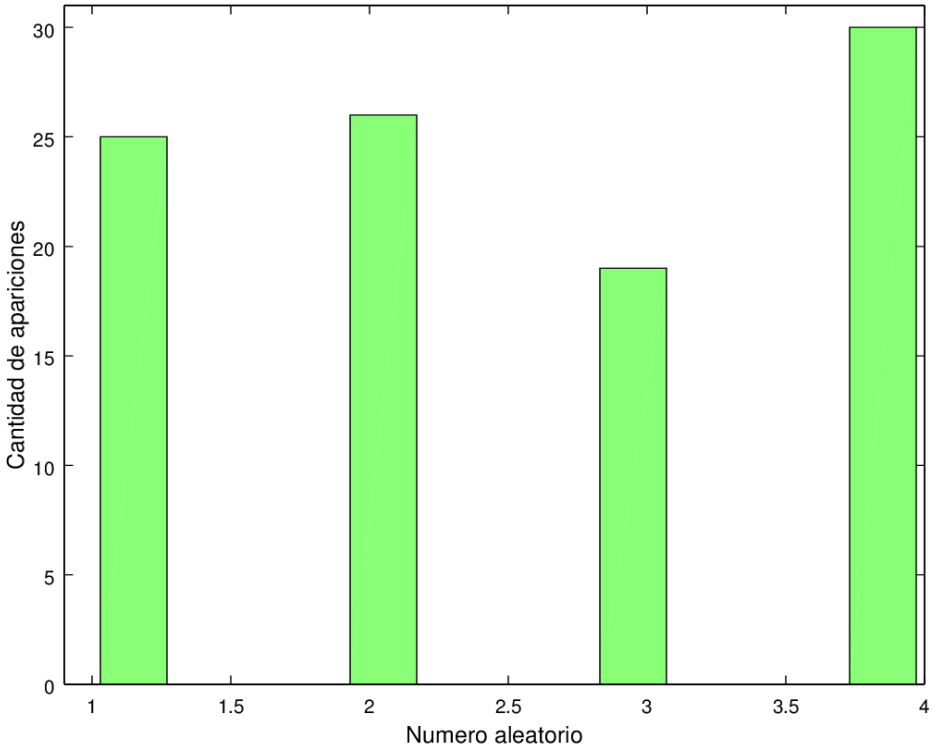

Figura 6: Histograma

3. Ejercicio 3

TaskBatch(int pid, vector(int) params)

```
1  int total_cpu ← params[0] - 1
2  int cant_bloqueos ← params[1]
3  int r
4  vector(int) randoms
5  r ← random(1, total_cpu)
6  randoms.push_back(r)
7  para (j = 1; j ≤ cant_bloqueos; j++) hacer
8  |  r = random(1, total_cpu)
9  |  mientras (estaEnElVector(r, randoms)) hacer
10 |  |  r ← random(1, total_cpu)
11 |  fin mientras
12 |  randoms.push_back(r)
13 fin
14 ordenar(randoms)
```

Para implementar esta tarea se crea un valor random entre 1 y el tiempo de cpu que queda disponible menos la cantidad de bloqueos restantes. Este número significará cuanto tiempo de cpu se utilizará hasta la siguiente llamada bloqueante. Solo puede ser entre esos valores ya que entre cada par de llamadas bloqueantes deberá utilizarse la cpu por al menos una unidad de tiempo. Como la cantidad de tiempo de procesamiento es limitada deberá tenerse en cuenta la cantidad mínima de tiempo necesaria a utilizar luego. Este proceso se repite tantas veces como llamadas bloqueantes tenga la función actualizando cada vez el tiempo de cpu y las llamadas bloqueantes. Cuando no quedan más llamadas por hacer, si quedaba tiempo de cpu se lo utiliza.

A continuación se presentan el lote utilizado y su correspondiente gráfico.

```
TaskBatch 20 2
@5:
TaskBatch 12 1
@7:
TaskBatch 25 4
TaskBatch 20 6
```

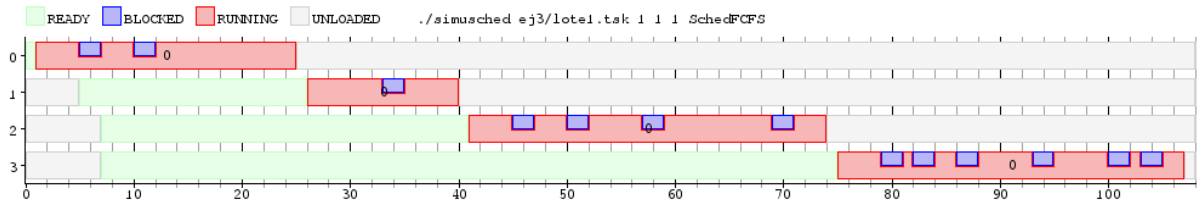

Figura 7: Scheduler: FCFS

4. Ejercicio 4

A continuación presentaremos un pseudocódigo de la implementación del Round-Robin. La idea general es tener una cola (cola) donde se irán encolando los procesos que estén en estado ready e ir desencolándolos a medida que se los quiera ejecutar o eventualmente bloquear. Además se utiliza un vector(quantum) en donde se guardan los quantums de cada procesador y otro vector (quantumRestante) donde se guarda el quantum restante al proceso en ejecución en un determinado procesador.

ROUND-ROBIN(vector<int> argn)

```
                ▷ Round-robin recibe la cantidad de cores y sus cpu_quantum por parámetro
1 para (i = 0; i < argn[0]; i++) hacer
2 |   quantum.push_back(argn[i+1])
                                          ▷ guardo los quantums de cada core
3 |   quantumRestante.push_back(argn[i+1])
                                  ▷ inicializo el quantumRestante de cada core
4 fin
```

LOAD(int pid)

```
1 cola.push(pid)
                                  ▷ si llega un nuevo proceso lo encolo
```

UNBLOCK(int pid)

```
1 cola.push(pid)
                                  ▷ si se desbloquea un proceso lo encolo
```

```
TICK(int pid, enum Motivo m) → res : int
1  int r ← IDLE_TASK
                                          ▷ analizo los casos de que sea exit o block
2  si (m == EXIT || m ==BLOCK) entonces
   |                                                ▷ reseteo el quantum vigente:
3  |  quantumRestante[cpu] ← quantum[cpu] - 1
   |                                  ▷ saco el proximo de la cola (o el idle si esta vacia):
4  |  si (!cola.empty()) entonces
5  |  |  r ← cola.front()
6  |  |  cola.pop()
7  |  en otro caso
8  |  |  r ← IDLE_TASK
9  |  fin si
10 en otro caso
   |                          ▷ si m==TICK, preguntar si se acabo el quantumRestante:
11 |  si (quantumRestante[cpu]== 0 || current_pid(cpu)==IDLE_TASK) entonces
12 |  |  quantumRestante[cpu] ← quantum[cpu] - 1
   |  |                          ▷ sacar el proximo si hay y encolar este, sino: sigo con este
13 |  |  si (!cola.empty()) entonces
14 |  |  |  r ← cola.front()
15 |  |  |  cola.pop()
16 |  |  |  si (current_pid(cpu)!=IDLE_TASK) entonces
17 |  |  |  |  cola.push(current_pid(cpu))
18 |  |  |  fin si
19 |  |  en otro caso
20 |  |  |  r ← current_pid(cpu)
21 |  |  fin si
22 |  en otro caso
23 |  |  quantumRestante[cpu]−−
24 |  |  r ← current_pid(cpu)
25 |  fin si
26 fin si
27 return r
```

Ejemplos:

```
TaskConsola 5 2 5
@3:
TaskConsola 2 1 6
@4:
*2 TaskConsola 3 1 3
```

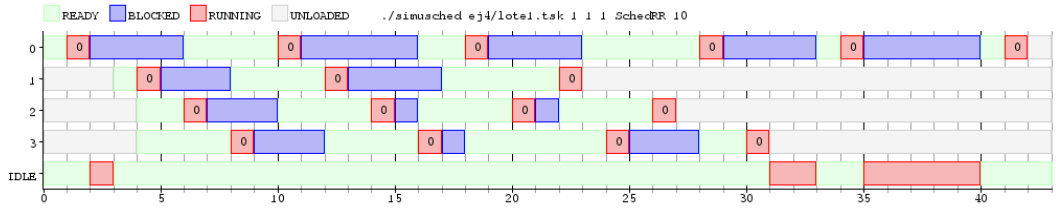

Figura 8: Scheduler: Round-Robin

```
TaskBatch 20 2
@5:
TaskBatch 12 1
@7:
TaskBatch 25 4
TaskBatch 20 6
```

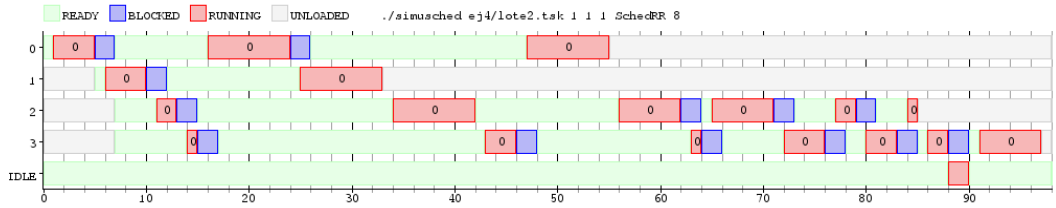

Figura 9: Scheduler: Round-Robin

5. Ejercicio 5

El lote utilizado fue el siguiente:

```
TaskCPU 70
TaskCPU 70
TaskCPU 70
TaskConsola 3 3 3
TaskConsola 3 3 3
TaskConsola 3 3 3
```

A continuación se presentan los gráficos correspondientes:

Figura 10: Quantum = 2

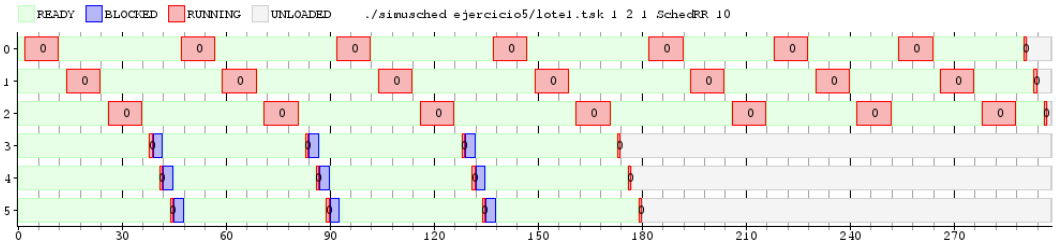

Figura 11: Quantum = 10

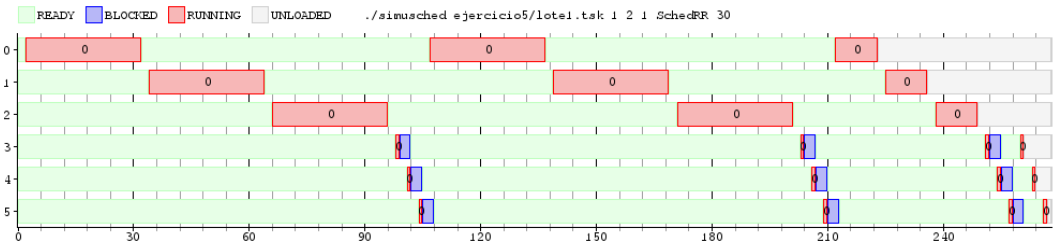

Figura 12: Quantum = 30

5.1. Waiting Time

Cuadro 2: Waiting Time

Tarea	**Quantum = 2**	**Quantum = 10**	**Quantum = 30**
0	459 - 71 - 72 = 316	291 - 71 - 16 = 204	223 - 71 - 6 = 146
1	462 - 71 - 72 = 319	294 - 71 - 16 = 207	236 - 71 - 6 = 159
2	465 - 71 - 72 = 322	297 - 71 - 16 = 210	249 - 71 - 6 = 172
3	78 - 4 -8 -9 = 57	174 - 4 - 8 - 9 = 153	261 - 4 - 8 - 9 = 240
4	81 - 4 - 8 - 9 = 60	177 - 4 - 8 - 9 = 156	264 - 4 - 8 - 9 = 243
5	84 - 4 - 8 - 9 = 63	180 - 4 - 8 - 9 = 159	267 - 4 - 8 - 9 = 246
Promedio	189,5	181,5	201

El waiting time de cada tarea se calculó realizando la siguiente cuenta:
Waiting Time = Tiempo Final - Tiempo de ejecución - Tiempo de Bloqueo total - 2 × Cantidad de cambios de contexto realizados
Donde tiempo final es el tiempo en el que termina de ejecutarse la tarea. Tiempo de ejecución es el tiempo total que la tarea utilizó la CPU.
Como se puede ver en la tabla el mayor waiting time se da cuando el quantum es 30. Esto se debe a que al dar mucho tiempo de ejecución para cada tarea, las que no están siendo ejecutadas deberán esperar esa misma cantidad de tiempo generando así un waiting time mayor.
A pesar de que el promedio es menor cuando el quantum es igual a 10, podemos notar que las tareas que realizan llamadas bloqueantes (3, 4 y 5) son beneficiadas cuando el quantum es igual a 2, ya que no utilizan demasiado la CPU. Contrariamente, las tareas que utilizan por 70 ciclos la CPU son perjudicadas, ya que el tiempo de cambio de contexto es el mismo que el quantum que se les otorga.

5.2. Latencia

Cuadro 3: Latencia

Tareas	**Quantum = 2**	**Quantum = 10**	**Quantum = 30**
0	0	0	0
1	4	12	32
2	8	24	64
3	12	36	96
4	15	39	99
5	18	42	102
Promedio	9,5	25,5	65,5

Como se puede ver en la tabla, a medida que se aumenta el quantum aumenta la latencia promedio. La latencia es el tiempo que pasa entre que llega una tarea y que es ejecutada por primera vez, y todas las tareas llegan en el tiempo 0. Por ello, las tareas deberán esperar para ser ejecutadas por primera vez el quantum × cantidad de tareas que se ejecutan antes que ella. Entonces si aumentamos el valor del quantum, aumentará la latencia de todas las tareas generando una latencia promedio mayor.

5.3. Tiempo total de ejecución

Cuadro 4: Tiempo total de ejecución

Tarea	Quantum = 2	Quantum = 10	Quantum = 30
0	459	291	223
1	462	294	236
2	465	297	249
3	78	174	261
4	81	177	264
5	84	180	267
Promedio	271,5	235,5	250
Tiempo Total	465	297	267

Como se puede ver en la tabla, para las tareas que se bloquean, lo más eficiente es cuando el quantum es 2, en cambio para las que no se bloquean es cuando el quantum es 30. Podemos notar que el mejor promedio se da con el quantum igual a 10 ya que este es un punto medio entre ambos valores.
Podemos notar además que el tiempo total que requiere ejecutar todas las tareas es mucho mayor cuando el quantum es 2. Esto se debe a que el tiempo que toma realizar un cambio de contexto es igual al que se destina a ejecutar la tarea. Por lo tanto, aproximadamente la mitad del tiempo total de ejecución se pierde realizando cambios de contexto. En cambio, cuando el quantum es 30, lo que se pierde haciendo cambios de contexto es despreciable en relacion al tiempo que se destina a la ejecución.

6. Ejercicio 6

El lote utilizado es el mismo lote que en el ejercicio 5:

```
TaskCPU 70
TaskCPU 70
TaskCPU 70
TaskConsola 3 3 3
TaskConsola 3 3 3
TaskConsola 3 3 3
```

Figura 13: Scheduler: FCFS

Para observar las diferencias entre el scheduler Round-Robin y FCFS, se calculó la latencia, waiting time y tiempo de ejecución total para este último scheduler:

Cuadro 5

Tarea	**Waiting Time**	**Latencia**	**Tiempo de ejecución total**
0	73 - 71 - 2 = 0	0	73
1	146 - 71 - 2 = 73	73	146
2	219 - 71 - 2 = 146	146	219
3	234 - 4 - 9 - 2 = 219	219	234
4	249 - 4 - 9 - 2 = 234	234	249
5	264 - 4 - 9 - 2 = 249	249	264
Promedio	153,5	153,5	169,28

Como se puede ver en la tabla, el promedio de los tiempos de ejecución en el caso del scheduler FCFS es menor a cualquiera de los promedios con los diferentes valores de quantum de Round-Robin. Esto se debe a que en FCFS hay una menor cantidad de cambios de contexto ya que una vez que comienza a ser ejecutada una tarea no se pasará a otra hasta que la misma termine.
Por este mismo motivo en FCFS la latencia promedio es mucho mayor. Solo la primer tarea tendrá latencia nula pero todo el resto deberá esperar la suma de los tiempos totales que duran las tareas que se ejecutarán antes que la misma.
Además, podemos notar que los valores de waiting time cuando se ejecuta con el scheduler FCFS son muy diferentes, ya que la primer tarea que llega no debe esperar nada, pero la última tiene que esperar que se terminen de ejecutar todas las otras tareas. En cambio, cuando se ejecuta con el scheduler en Round-Robin, los valores de waiting time por tarea son similares, debido a que al cambiar de tarea constantemente en ningún momento sucede que hay una tarea esperando un largo plazo de tiempo. El tiempo promedio de waiting time en el scheduler de Round-Robin se ve incrementado también porque para todas las tareas que no sean la que esta por utilizar la CPU el cambio de contexto les suma waiting time. Al haber mayor cantidad de cambios de contexto el waiting time de cada tarea se incrementará generando así mayor waiting time promedio.

7. Ejercicio 7

El scheduler Mistery es un scheduler de tipo Round-Robin con colas de prioridad. El mismo recibirá por parámetro una cantidad de valores que representan los quantums de cada cola siendo el primer valor el correspondiente a la cola de mayor prioridad entre las pasadas por parámetro y las prioridades van disminuyendo hasta el último. Además se agregara como cola de máxima prioridad una de quantum = 1. Cada vez que se esta ejecutando una tarea y se acaba el quantum de la misma sin que ésta se bloquee, la tarea pasará a la cola de prioridad que se encuentra a la derecha, es decir, la que tiene exactamente un grado de prioridad menos. En caso de que la misma se bloquee antes de que finalice el quantum, ésta pasará a la cola de prioridad de la izquierda, es decir, la que tiene exactamente un grado mas de prioridad. Si se encuentra en la cola de máxima prioridad y se bloquea antes de que se termine el quantum la tarea no cambiará de cola. Lo mismo sucede si se encuentra en la cola de mínima prioridad y se termina el tiempo. Las tareas llegan y se encolan en la cola de máxima prioridad. Cada vez que se acabe el quantum de algún proceso, el próximo proceso será el primero de la cola de prioridad más

alta(el de más a la izquierda).

```
TaskCPU 30
TaskCPU 20
TaskCPU 10
```

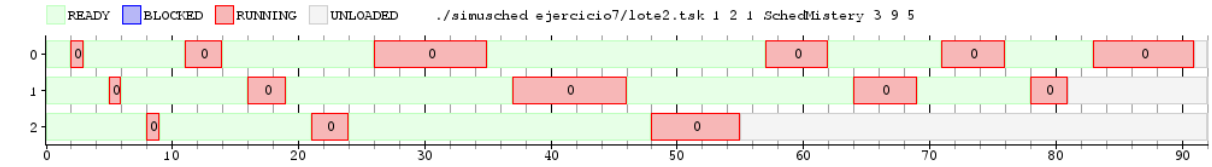

Figura 14: Lote 1

```
TaskAlterno 10 2 20 4
TaskAlterno 3 10 15 6 10
TaskCPU 30
```

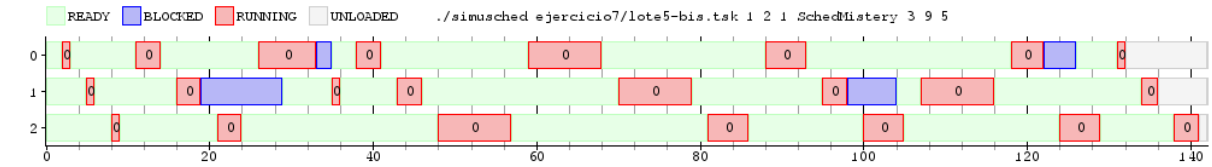

Figura 15: Lote 2

```
TaskAlterno 6 10 17 5 21 1 21
TaskAlterno 10 5 10
```

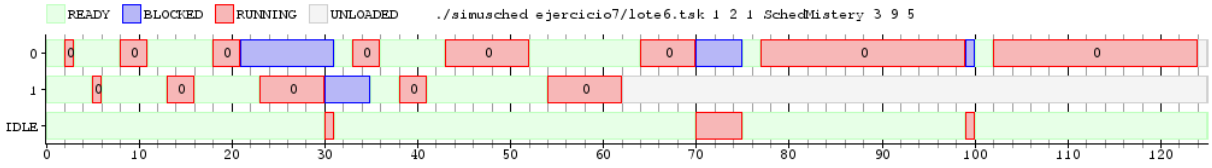

Figura 16: Lote 3

Con el lote 1 se puede ver que son colas de prioridad cuyos quantums son los pasados por parámetro donde la de mayor prioridad se corresponde con el primer valor pasado y va disminuyendo hasta el último. Se puede ver también que primero se ejecuta un solo tiempo. Se puede observar que una vez que llegó a la cola de mas a la derecha (en este caso la de quantum = 5) se queda en esa cola si se le vuelve a acabar el tiempo.

En el lote 2, en la tarea 1 se puede ver como después de bloquearse, como no había terminado el quantum, pasa de nuevo a ejecutar solo un tiempo por lo que tiene que haber una cola de máxima prioridad con quantum 1. En la tarea 2 se puede ver que ejecuta las colas en orden ya que siempre se le acaba el tiempo. En la tarea 0 pasa lo mismo después del primer bloqueo.

En el lote 3 se puede ver que si se bloquean y todavía no se había terminado el quantum, las tareas vuelven a la cola de la izquierda y si no se les había terminado van a la de la derecha.

8. Ejercicio 8

La implementación pedida en este ejercicio esta fuertemente basada en la implementación del ejercicio 4 con la salvedad de que se utiliza una cola por cada procesador(almacenadas en colas), un diccionario o map que en el que se guarda a que procesador pertenece cada proceso (pid - cpu) para recordar cuando un proceso vuelve del bloqueo a que procesador pertenecía y contador de procesos(cantProc) que cuenta cuantos procesos en total tiene cada procesador y saber así a cual procesador asignar un nuevo proceso.

SchedRR2(vector< *int* > argn)

```
                    ▷ Round-robin recibe la cantidad de cores y sus cpu_quantum por parámetro
1  queue< int > cola;
                      ▷ por cada core asigno su quantum, seteoquantumRestante y creo su cola:
2  para (int i = 0; i < argn[0]; i++) hacer
3  |   quantum.push_back(argn[i+1])
4  |   quantumRestante.push_back(argn[i+1])
5  |   colas.push_back(cola)
6  |   cantProc.push_back(0)
7  fin
8  cantCores ← argn[0]
```

load(int pid)

```
                                                    ▷ busco la cpu con menos procesos:
1  int proxCpu ← next()
                                                  ▷ encolo en la cola de dicho procesador:
2  colas[proxCpu].push(pid)
                                                ▷ Mapeo el nuevo proceso al procesador:
3  pid_cpu.insert( pid,proxCpu)
                                                      ▷ incremento su cantidad de procesos
4  cantProc[proxCpu]++
```

UNBLOCK(int pid)

▷ obtengo el procesador al cual correspondia el proceso y encolo:

```
1 int cpu ← pid_cpu.at(pid)
2 colas[cpu].push(pid)
```

TICK(int *cpu*, const enum Motivo m) → res : `int`

▷ Realizo el algoritmo analogo al explicado en detalle en el Ej.4 solo que teniendo en cuenta la variedad de colas y de que eventualmente cuando un proceso termine,decremento la cantidad de procesos de cada procesador y borro la entrada del diccionario con clave=pid del proceso que termina:

```
1 int r ← IDLE_TASK
2 si (m == EXIT || m == BLOCK) entonces
3 |  quantumRestante[cpu] ← quantum[cpu]-1
4 |  si (!colas[cpu].empty()) entonces
5 |  |  r = colas[cpu].front()
6 |  |  colas[cpu].pop()
7 |  en otro caso
8 |  |  r = IDLE_TASK
9 |  fin si
10|  si (m == EXIT) entonces
11|  |  pid_cpu.erase(current_pid(cpu))
12|  |  cantProc[cpu]–
13|  fin si
14 en otro caso
15|  si (quantumRestante[cpu] == 0 || current_pid(cpu)==IDLE_TASK) entonces
16|  |  quantumRestante[cpu] ← quantum[cpu]-1
17|  |  si (!colas[cpu].empty() ) entonces
18|  |  |  r = colas[cpu].front()
19|  |  |  colas[cpu].pop()
20|  |  |  si (current_pid(cpu)!=IDLE_TASK) entonces
21|  |  |  |  colas[cpu].push(current_pid(cpu) )
22|  |  |  fin si
23|  |  en otro caso
24|  |  |  r ← current_pid(cpu)
25|  |  fin si
26|  en otro caso
27|  |  quantumRestante[cpu]−−
28|  |  r ← current_pid(cpu)
29|  fin si
30 fin si
31 return r
```

```
NEXT() → res : int
                                          ▷ busco el cpu con la menor cantidad procesos:
1 int cpuMinimo ← 0
2 para (int i = 0; i < cantCores; i++) hacer
3 |   si (cantProc[i] < cantProc[cpuMinimo]) entonces
4 |   |   cpuMinimo ← i
5 |   fin si
6 fin
7 return cpuMinimo
```

Un caso real donde la migración de núcleos es beneficiosa es, por ejemplo, cuando entre el lote de tareas, se tienen tareas de corta duración y de larga duración alternadas. Si consideramos 2 núcleos, las tareas de corta duración van a cargarse en uno de ellos, y las de larga duración en el otro. Por lo tanto, uno de los procesadores terminará de ejecutar todas sus tareas rápidamente, mientras que en el otro hay tareas que todavía no empezaron a ejecutarse. En este caso, al liberarse el primer procesador, se podrá migrar tareas del otro, y así mejorar el rendimiento. Si se compara el tiempo de ejecución total promedio cuando hay cambio de núcleo y cuando no hay, se podrá notar que con cambio de procesador el resultado será menor. Esto se debe a que al aprovechar los dos procesadores todo el tiempo, no hay uno que no este haciendo nada mientras que hay tareas que no empezaron a ejecutarse. También, el valor de latencia promedio podría verse disminuido cuando hay migración, ya que las tareas de larga duración que se corran en el segundo procesador, podrían empezar antes de lo que hubieran empezado si se ejecutaban en el primero. Por este mismo motivo también disminuye el waiting time promedio.
Por otro lado, un caso donde la migración de núcleos no es beneficiosa es cuando el tiempo que requiere cambiar una tarea de procesador es muy significativo en comparación al tiempo que dura una tarea y al tiempo que tarda en pasar de una tarea a otra en el mismo procesador. En este caso si no hay cambio de procesador, entonces la latencia podría verse disminuida ya que el tiempo de espera a que el procesador termine de procesar la tarea que se está ejecutando es menor al tiempo que requiere cambiarse de núcleo.

8.1. Lote 1

```
TaskCPU 2
TaskCPU 10
TaskCPU 2
TaskCPU 10
TaskCPU 2
TaskCPU 10
```

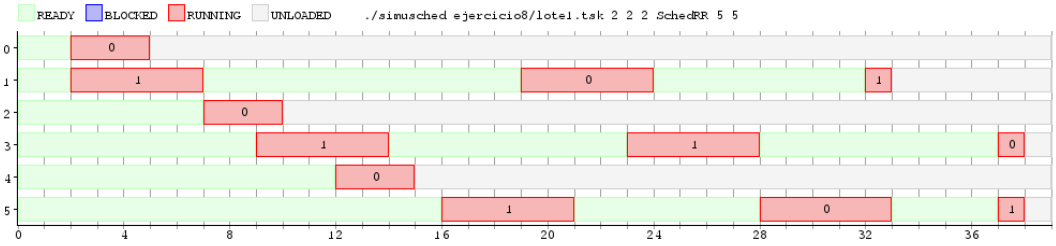

Figura 17: Scheduler: RR

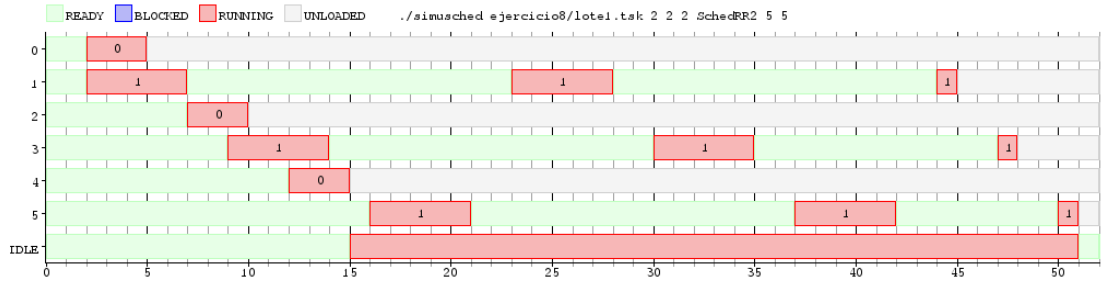

Figura 18: Scheduler: RR2

Cuadro 6: Lote 1

Promedio	**con migración entre núcleos**	**sin migración entre núcleos**
Latencia	6	6
Waiting Time	10.5	18
Tiempo total de ejecución	23.16	29

Se puede observar en el cuadro 6 que los valores calculados se corresponden con lo explicado anteriormente. El waiting time y tiempo de ejecución total mejoran cuando hay migración entre núcleos.

8.2. Lote 2

```
TaskCPU 6
TaskCPU 8
TaskCPU 7
TaskCPU 7
TaskCPU 9
TaskCPU 8
```

Figura 19: Scheduler: RR

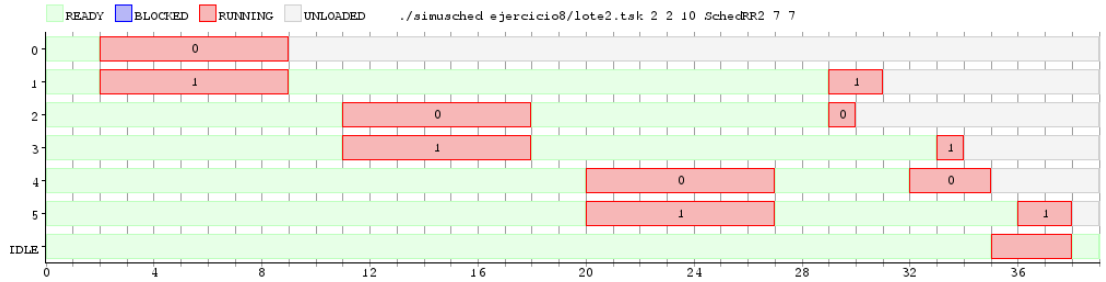

Figura 20: Scheduler: RR2

Cuadro 7: Lote 2

Promedio	**con migración entre núcleos**	**sin migración entre núcleos**
Latencia	11	11
Waiting Time	22,16	17,3
Tiempo total de ejecución	37,6	29,5

Se puede observar en el cuadro 7 que los valores calculados se corresponden con lo explicado anteriormente. El waiting time y tiempo de ejecución total mejoran cuando no hay migración entre núcleos.