DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Scheduling

Sistemas Operativos

Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Alem Santiago	650/14	santialem.trev@gmail.com
Alliani Federico	183/15	fedealliani@gmail.com
Raposeiras Lucas	034/15	lucas.raposeiras@outlook.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

Resumen

El objetivo de este Trabajo Práctico es aprender el funcionamiento de algunos tipos de scheduler, analizar las ventajas y desventajas de cada uno de ellos y entender algunos conceptos que los distinguen unos de otros, como la *latencia*, *waiting time*, *throughput*, *quantum*, etc.

En este informe explicaremos cómo realizamos cada ejercicio del Trabajo Práctico, así como también algunas conclusiones obtenidas.

Índice

1. Ejercicio 1

Tipo de tarea: `TaskConsola` *n bmin bmax*

La tarea `TaskConsola` consiste en realizar *n* llamadas bloqueantes, donde *n* es el primer parámetro de la tarea. Cada llamada bloqueante debe tener una duración al azar entre dos valores mínimo y máximo, ambos especificados en el segundo y tercer parámetro respectivamente.

Implementación

Para implementar este tipo de tarea, utilizamos la función `void uso_IO(int pid, unsigned int ms)` que la cátedra provee. Esta función realiza una llamada bloqueante, y recibe como parámetros el `pid` de la tarea y la cantidad de ciclos que durará el bloqueo. Dicha función es llamada con una duración al azar[1] entre *bmin* y *bmax*, en un ciclo de *n* repeticiones.

Para poder usar la nueva tarea en cualquiera de los scheduler implementados debemos registrarla usando la *macro* `register_task` que provee la cátedra.

Lote y gráficos

Utilizaremos el siguiente lote y observaremos el gráfico resultante para certificar el comportamiento de la nueva tarea.

```
1 TaskConsola 3 5 9
2 @10:
3 TaskConsola 2 2 5
4 @3:
5 TaskConsola 5 1 3
```

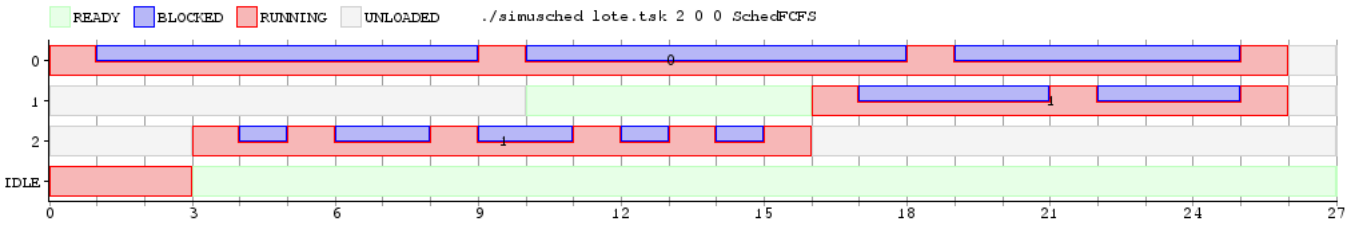

Figura 1: Gráfico del lote propuesto utilizando el scheduler *FCFS*

Se puede ver en el gráfico de la fig. 1 cómo cada tarea se bloquea durante una duración al azar entre los parámetros utilizados en el lote, y luego de hacer *n* llamadas bloqueantes se termina la ejecución de la misma.

2. Ejercicio 2

Latencia, waiting time y throughput

Latencia: La *latencia* es el tiempo desde que una tarea es cargada en el scheduler hasta que hasta que se ejecuta por primera vez.

Waiting time: El *waiting time* es la sumatoria de los tiempos en los que la tarea se encuentra en estado ready.

Throughput: El *throughput* es la cantidad de tareas ejecutadas por el scheduler dividido el tiempo total de ejecucion de todas las tareas. Esto nos da una medida de la cantidad de tareas por unidad de tiempo que se ejecutan.

[1] La duración al azar la obtenemos con la función `randomRange`, implementada por nosotros, que a su vez utiliza la función `rand()` de la librería `stdlib.h` y devuelve un número al azar entre *bmin* y *bmax*.

Lote a analizar

Para este ejercicio utilizaremos el siguiente lote propuesto por la cátedra.

```
TaskCPU 10
@5:
TaskConsola 5 1 3
@6:
TaskConsola 5 1 4
@8:
TaskCPU 8
```

Gráficos y resultados

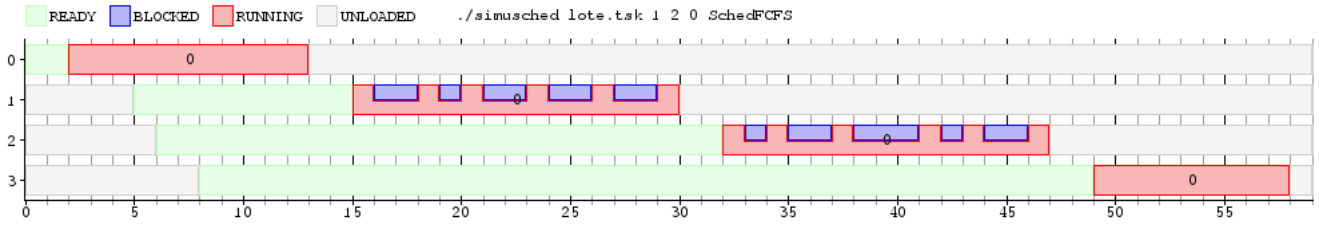

Figura 2: Gráfico del lote propuesto utilizando el scheduler *FCFS* para 1 núcleo y un cambio de contexto de 2 ciclos.

Tarea	*Latencia*	*Waiting time*
Tarea 0	2	2
Tarea 1	10	10
Tarea 2	26	26
Tarea 3	41	41

Cuadro 1: Tabla de valores de *latencia* y *waiting time* de cada tarea del lote propuesto utilizando el scheduler *FCFS* para 1 núcleo y un cambio de contexto de 2 ciclos.

$$Throughput = \frac{4}{58} = 0,0689655$$

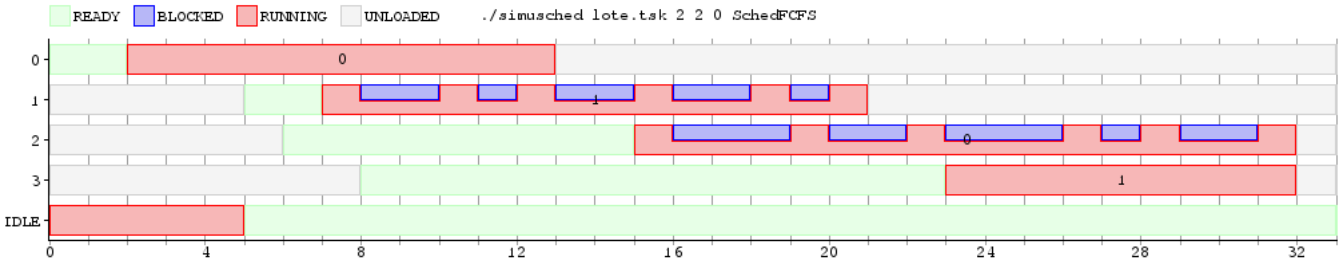

Figura 3: Gráfico del lote propuesto utilizando el scheduler *FCFS* para 2 núcleos y un cambio de contexto de 2 ciclos.

Tarea	*Latencia*	*Waiting time*
Tarea 0	2	2
Tarea 1	2	2
Tarea 2	9	9
Tarea 3	15	15

Cuadro 2: Tabla de valores de *latencia* y *waiting time* de cada tarea del lote propuesto utilizando el scheduler *FCFS* para 2 núcleos y un cambio de contexto de 2 ciclos.

$$Throughput = \frac{4}{32} = \frac{1}{8} = 0,125$$

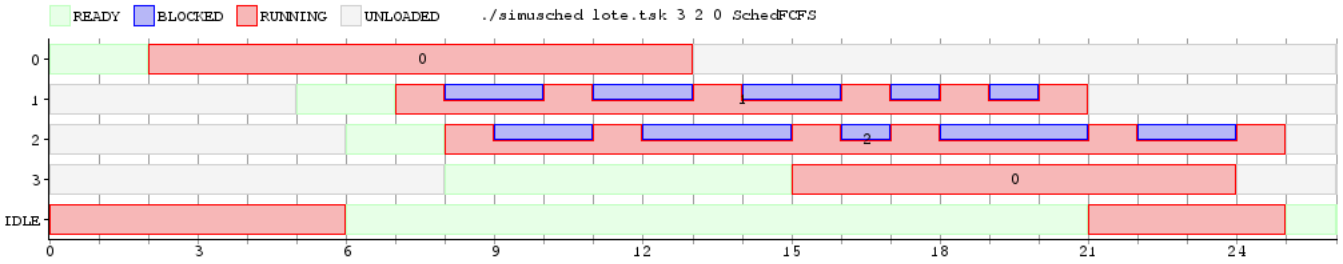

Figura 4: Gráfico del lote propuesto utilizando el scheduler *FCFS* para 3 núcleos y un cambio de contexto de 2 ciclos.

Tarea	*Latencia*	*Waiting time*
Tarea 0	2	2
Tarea 1	2	2
Tarea 2	2	2
Tarea 3	7	7

Cuadro 3: Tabla de valores de *latencia* y *waiting time* de cada tarea del lote propuesto utilizando el scheduler *FCFS* para 3 núcleos y un cambio de contexto de 2 ciclos.

$$Throughput = \frac{4}{25} = 0,16$$

3. Ejercicio 3

Tipo de tarea: `TaskPajarillo` *cantidad_repeticiones tiempo_cpu tiempo_bloqueo*

La tarea `TaskPajarillo` consiste en realizar *n* llamadas bloqueantes y *n* usos de CPU (*n* es un parámetro) con las duraciones especificadas por parámetro.

Implementación

Para implementar este tipo de tarea, utilizamos las funciones `void uso_IO(int pid, unsigned int ms)` y `void uso_CPU(int pid, unsigned int ms)` que la cátedra provee. La función `uso_IO` realiza una llamada bloqueante, y recibe como parámetros el `pid` de la tarea y la cantidad de ciclos que durará el bloqueo, mientras que la función `uso_CPU` hace uso del cpu durante una cantidad de ciclos especificada por parámetro.

La implementación la realizamos a partir de un ciclo de *n* repeticiones el cual se encarga de llamar a las funciones `uso_CPU` y `uso_IO`, con los parámetros respectivos especificados por la tarea.

Finalmente, para poder usar la nueva tarea en cualquiera de los scheduler implementados debemos registrarla usando la *macro* `register_task` que provee la cátedra.

Lote y gráficos

Utilizaremos el siguiente lote y observaremos el gráfico resultante para certificar el comportamiento de la nueva tarea.

```
TaskPajarillo 4 3 5
@2:
TaskPajarillo 1 2 4
@3:
TaskPajarillo 5 4 2
```

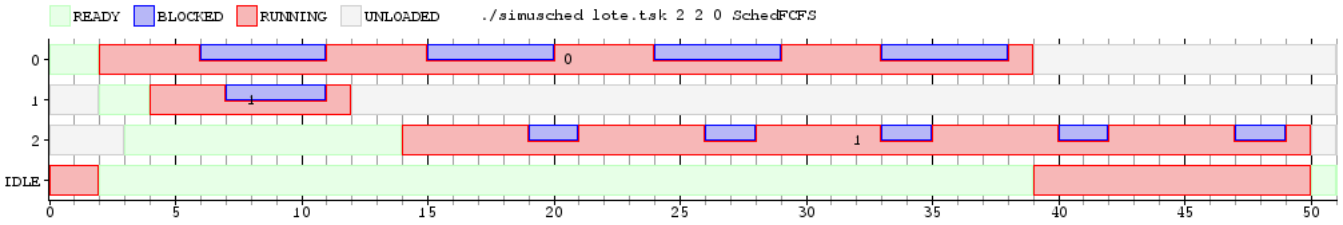

Figura 5: Gráfico del lote propuesto utilizando el scheduler *FCFS* para 2 núcleos y un cambio de contexto de 2 ciclos.

Tarea	***Latencia***	***Waiting time***
Tarea 0	2	2
Tarea 1	2	2
Tarea 2	11	11
Promedio	5	5

Cuadro 4: Tabla de valores de *latencia* y *waiting time* de cada tarea y el promedio del lote propuesto utilizando el scheduler *FCFS* para 2 núcleos y un cambio de contexto de 2 ciclos.

$$Throughput = \frac{3}{50} = 0,06$$

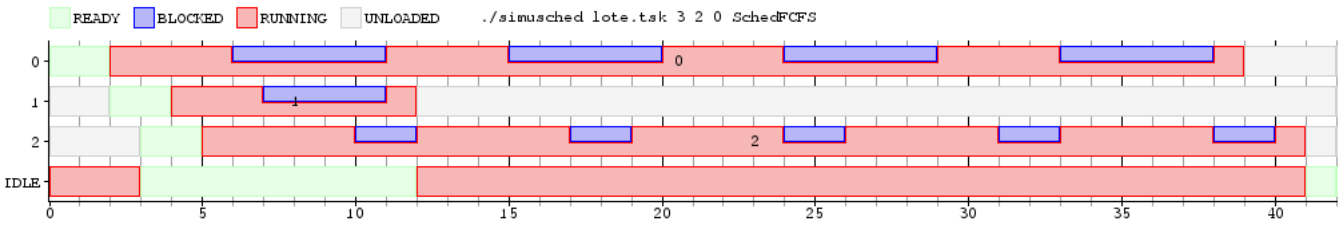

Figura 6: Gráfico del lote propuesto utilizando el scheduler *FCFS* para 3 núcleos y un cambio de contexto de 2 ciclos.

Tarea	*Latencia*	*Waiting time*
Tarea 0	2	2
Tarea 1	2	2
Tarea 2	2	2
Promedio	2	2

Cuadro 5: Tabla de valores de *latencia* y *waiting time* de cada tarea y el promedio del lote propuesto utilizando el scheduler *FCFS* para 3 núcleos y un cambio de contexto de 2 ciclos.

$$\textit{Throughput} = \frac{3}{41} = 0,\widehat{07317}$$

4. Ejercicio 4

Funcionamiento del scheduler *Round-Robin*

La política general del scheduler *Round-Robin* consiste en alternar la ejecución de los procesos que estén en estado *ready* cada cierta cantidad de *quantums*, sin importar que estos no hayan terminado su ejecución. Además, en esta implementación, cuando una tarea se bloquea, la misma se desaloja y es asignada la siguiente tarea en estado *ready*.

Implementación

Las tareas que están en estado *ready* serán guardadas por orden de llegada en una cola sin prioridad llamada `tasksReady`. Por otro lado, las tareas que estén bloqueadas, serán guardadas temporalmente en la cola `tasksBloqueadas`[1].

Por otro lado, definimos la estructura `cpu_data`, la cual contiene la información de la tarea actual, *quantums* transcurridos en la ejecución actual y *quantums* totales asignados al CPU. Almacenamos la información de cada CPU en el vector `cpus`.

La implementación del constructor del scheduler consiste en inicializar el vector `cpus`, creando un elemento en el vector por cada CPU especificado en los parámetros del scheduler. Para cada CPU la tarea inicial es `IDLE_TASK`[1], los *quantums* transcurridos son 0, y los *quantums* para alternar las tareas es definido según los valores recibidos en los parámetros del scheduler.

En la función `void SchedRR::load(int pid)` únicamente encolamos la tarea recibida por parámetro en la cola de tareas que están en estado *ready*. Esta función es llamada por el simulador implementado por la cátedra cuando una tarea pasa a estado *ready* por primera vez.

En la función `void SchedRR::unblock(int pid)` buscamos el `pid` en la cola de tareas bloqueadas y la movemos al final de la cola de tareas en estado *ready*.

La funcion `int SchedRR::tick(int cpu, const enum Motivo m)` es llamada por simulador luego de que el CPU especificado realiza un tick. La misma tiene como parámetro el motivo del *tick*, que representa lo ocurrido durante el último *tick* ejecutado en la tarea actual del CPU especificado. Si el motivo especificado es que la tarea finalizó entonces procedemos a *resetear* los *quantums* transcurridos en el CPU y luego le asignamos al CPU la primer tarea de la cola de tareas en estado *ready*, y en caso de que no haya ninguna, se le asigna la tarea `IDLE_TASK`. Si el motivo especificado es que la tarea se bloqueó, se procede a mover dicha tarea a la cola de tareas bloqueadas, procedemos a *resetear* los *quantums* transcurridos en el CPU y luego le asignamos al CPU la primer tarea de la cola de tareas en estado *ready*, y en caso de que no haya ninguna, se le asigna la tarea `IDLE_TASK`. Si el motivo especificado no es ninguno de los anteriores, incrementamos los *quantums* transcurridos del CPU y verificamos si la tarea ya cumplió la cantidad de *quantums* totales a ejecutar. Si ya transcurrieron sus *quantums*, se mueve la tarea del CPU a la cola de tareas en estado *ready*, se *resetea* la cantidad de *quantums* transcurridos y luego se le asigna al CPU la primer tarea de la cola de tareas en estado *ready*, y en caso de que no haya ninguna, se le asigna la tarea `IDLE_TASK`. Si no transcurrieron todos los *quantums* de la tarea, y dicha tarea **no** es la tarea `IDLE_TASK`, se continúa con la ejecución de la misma. Si la tarea es la tarea `IDLE_TASK` se verifica si hay alguna tarea en estado *ready* para ser ejecutada.

[1] Si hay alguna tarea a comenzar en tiempo 0, esa tarea pisará a la tarea `IDLE_TASK`.

Para facilitar la implementación utilizamos dos funciones auxiliares, `void TaskSwitch(int cpu,const enum Motivo m)` e `int nextReadyTask()` que realizan lo antes explicado.

Lote y gráficos

Utilizaremos el siguiente lote y observaremos el gráfico resultante para certificar el comportamiento del nuevo scheduler.

```
TaskCPU 10
@2:
TaskIO 5 1
@4:
TaskConsola 5 1 4
@5:
TaskCPU 8
```

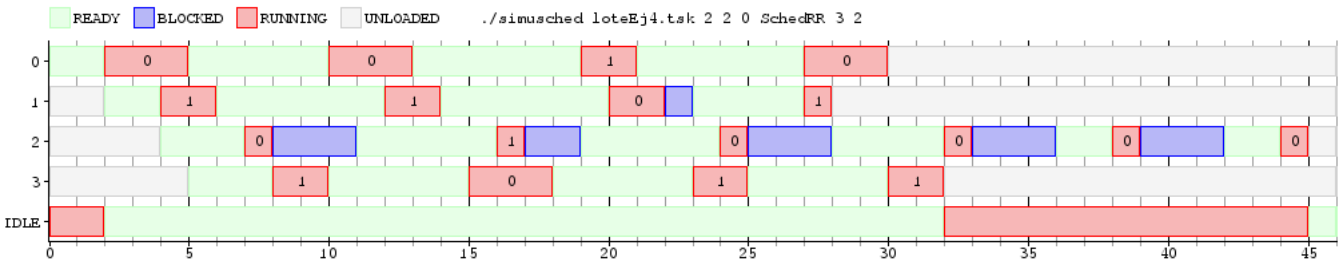

Figura 7: Gráfico del lote propuesto utilizando el scheduler *Round-Robin* para 2 núcleos, un cambio de contexto de 2 ciclos y 3 y 2 valores de *quantums* para el primer y segundo núcleo respectivamente.

Se puede ver en la fig. 7 que el scheduler se comporta de forma deseada para el lote dado.

5. Ejercicio 5

Características del scheduler *SchedMistery*

Para experimentar con este scheduler, utilizaremos los siguientes lotes[1].

Lote 1

```
TaskCPU 8
@2:
TaskCPU 5
TaskCPU 4
@3:
TaskCPU 3
TaskCPU 1
```

[1] El scheduler *SchedMistery* funciona sólo para tareas de tipo `TaskCPU`

Lote 2

```
TaskCPU 15
TaskCPU 20
@2:
TaskCPU 1
@3:
TaskCPU 2
@4:
TaskCPU 10
```

A continuación se muestran los gráficos de ejecución de los lotes 1 y 2 para 2 núcleos y un cambio de contexto de 1 ciclo y para 1 núcleo y sin costo de cambio de contexto respectivamente.

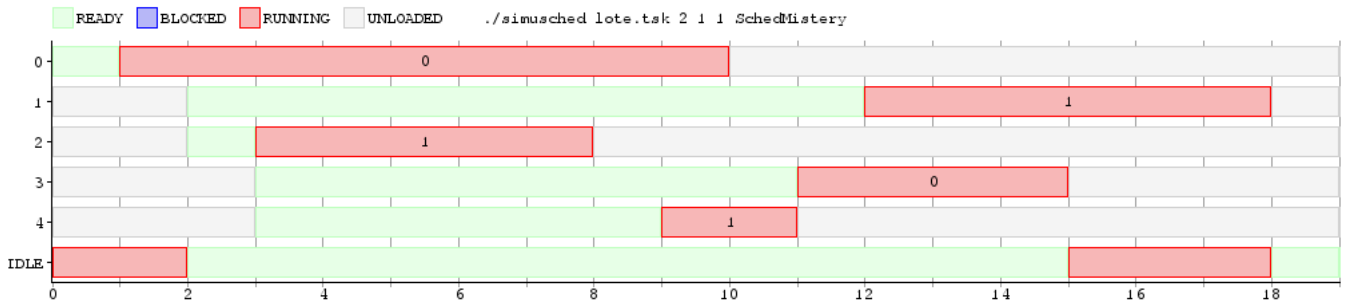

Figura 8: Gráfico del lote 1 utilizando el scheduler *SchedMistery* para 2 núcleos y un cambio de contexto de 1 ciclo.

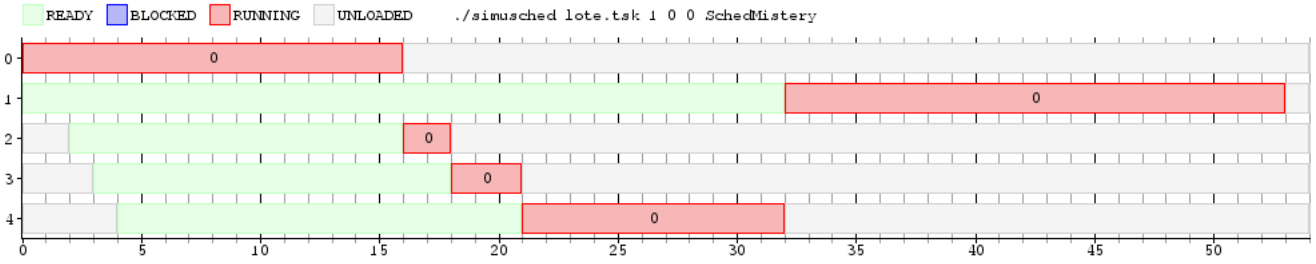

Figura 9: Gráfico del lote 2 utilizando el scheduler *SchedMistery* para 1 núcleo y sin costo de cambio de contexto.

Al analizar los gráficos de las fig. 8 y 9 notamos las siguientes observaciones:

- Para decidir cuál es la siguiente tarea a ejecutarse en un núcleo dado, el scheduler opta por la de menor duración.
- Una vez que una tarea inicia, la misma es ejecutada hasta que termina.

Luego de probar varios lotes similares se verificó que el scheduler *SchedMistery* utiliza la política de *SJF* entre las tareas en estado *ready*.

Implementación SchedNoMistery

Utilizamos un vector de tareas en estado *ready* llamado `tasksReady`[1]. Definimos una función auxiliar llamada `int getNextReadyAndDelete()` que elimina una tarea finalizada y retorna la próxima tarea a ejecutar.

[1] Utilizamos un vector y en lugar de una cola porque no nos importa el orden de llegada de las tareas, sólo su duración

En la implementación del constructor del scheduler no hace falta código, ya que no debemos inicializar ninguna estructura.

En la función `void SchedNoMistery::load(int pid)` únicamente agregamos la tarea recibida por parámetro al vector de tareas que están en estado *ready*. Esta función es llamada por el simulador implementado por la cátedra cuando una tarea pasa a estado *ready* por primera vez.

En la función `void SchedNoMistery::unblock(int pid)` no hace falta código ya que este scheduler únicamente funciona con tareas de tipo `TaskCPU`, y éstas no se bloquean nunca, con lo cual nunca se desbloquean, y esta función nunca es llamada por el simulador.

La funcion `int SchedNoMistery::tick(int cpu, const enum Motivo m)` es llamada por simulador luego de que el CPU especificado realiza un tick. La misma tiene como parámetro el motivo del *tick*, que representa lo ocurrido durante el último *tick* ejecutado en la tarea actual del CPU especificado. Si el motivo especificado es que la tarea finalizó y el vector de tareas en estado *ready* no está vacío, llamamos a la función `getNextReadyAndDelete` que nos devolverá la próxima tarea a ejecutar. Esta función busca en el vector de tareas en estado *ready* cuál es la tarea más corta (utilizando la función `tsk_params`, que nos devuelve una lista de parámetros y preguntando por la cantidad de ciclos que la tarea usa del CPU) y antes de retornarla, la elimina del vector. Si el vector de tareas en estado *ready* está vacío se ejecuta la tarea `IDLE_TASK`. Si el motivo es un *tick*, preguntamos si la tarea que se está ejecutando es la tarea `IDLE_TASK`. Si es la tarea `IDLE_TASK` y el vector de tareas no está vacío, llamamos a la función `getNextReadyAndDelete` antes mencionada. Si el vector está vacío o la tarea que se está ejecutando no es la tarea `IDLE_TASK` entonces se sigue ejecutando la tarea actual.

6. Ejercicio 6

Tipo de tarea: `TaskPriorizada` *prioridad tiempo_cpu*

La tarea `TaskPriorizada` consiste en hacer uso del CPU con la duración especificada en el segundo parámetro de la misma. El primer parámetro de la tarea, que corresponde a la prioridad de la misma, será utilizado por el scheduler *PSJF*.

Funcionamiento del scheduler *PSJF*

La política de este scheduler *PSJF* consiste en ejecutar primero las tareas más prioritarias y, dentro de las más prioritaras, hasta que se encuentre con otra tarea de mayor prioridad o con la misma prioridad pero más corta o finalize la ejecución de la misma.

Implementación

Para la implementación del scheduler *PSJF* utilizamos las siguientes estructuras: un arreglo de tamaño 5 de vectores llamado `tasksReady` que contienen las tareas en estado *ready* según su prioridad. Por ejemplo, si una tarea está en estado *ready* y tiene prioridad 4, se encontrará en el vector de la posicion 3 en arreglo `tasksReady`. Además creamos una estructura llamada `cpu_data_PSJF` que contiene dos `int` llamados `tareaActual` y `prioridad`. Esta estructura se utiliza para saber, en cada CPU, qué tarea se está ejecutando y qué prioridad tiene. Es por esto que tenemos un vector de structs `cpu_data_PSJF` llamado `cpus`.

Luego utilizamos dos funciones llamadas `getNextReady` y `getNextReadyAndDiscardActual`. La función `getNextReady` nos devuelve la tarea a ser ejecutada por el CPU especificado por parámetro en el próximo *tick*. La función *getNextReadyAndDiscardActual* nos devuelve la tarea a ser ejecutada por el CPU especificado por parámetro en el próximo *tick* y elimina la tarea actual que está siendo ejecutada actualmente en dicho CPU.

La implementación del constructor del scheduler consiste en inicializar el vector `cpus`, creando un elemento en el vector por cada CPU especificado en los parámetros del scheduler. Para cada CPU, la tarea inicial es `IDLE_TASK`[1] y la prioridad de la tarea actual será de 4 (ya que la tarea `IDLE_TASK` tiene la menor prioridad, en la escala del 0 a 4).

[1] Si hay alguna tarea a comenzar en tiempo 0, esa tarea pisará a la tarea `IDLE_TASK`

En la función `void SchedPSJF::load(int pid)` únicamente agregamos la tarea recibida por parámetro al vector de la posición del arreglo `tasksReady` que corresponda según su prioridad (para obtener su prioridad utilizamos la función `tsk_params`). Esta función (`SchedPSJF::load`) es llamada por el simulador implementado por la cátedra cuando una tarea pasa a estado *ready* por primera vez.

En la función `void SchedPSJF::unblock(int pid)` no hace falta ninguna línea de código ya que este scheduler únicamente acepta tareas del tipo `TaskPriorizadas`, y éstas no se bloquean nunca, con lo cual nunca se desbloquean, y esta función nunca es llamada por el simulador.

La función `int SchedPSJF::tick(int cpu, const enum Motivo m)` es llamada por el simulador luego de que el CPU especificado realiza un tick. La misma tiene como parámetro el motivo del *tick*, que representa lo ocurrido durante el último *tick* ejecutado en la tarea actual del CPU especificado. Si el motivo es que la tarea finalizó entonces llamamos a la función `getNextReadyAndDiscardActual`. Esta función busca en `tasksReady` la tarea con mayor prioridad, buscando primero en el vector de la posición 0 del arreglo, luego en el vector de la posición 1 del arreglo, etc. Si alguno de esos vectores no está vacío, como estamos buscando por orden creciente en cuanto a prioridades, se busca dentro del primer vector no vacío a la tarea más corta utilizando la función `tsk_params` para saber cuánto dura cada tarea. Una vez conseguida la tarea más prioritaria y más corta, la asignamos al CPU especificado por parámetro en la funcion `SchedPSJF::tick` y la tarea que se estaba ejecutando no se almacena en ningún lado, ya que la misma terminó. Si no hay tareas en ningún vector se asigna al CPU la tarea `IDLE_TASK` y la tarea que se estaba ejecutando no se almacena en ningún lado, ya que la misma terminó. Si el motivo es un *tick*, preguntamos si la tarea que se está ejecutando actualmente es la tarea `IDLE_TASK`. Si es la tarea `IDLE_TASK` entonces llamamos a la función `getNextReadyAndDiscardActual` explicada anteriormente. En cambio si no es la tarea `IDLE_TASK` entonces llamamos a la función `getNextReady`. La función `getNextReady`, al igual que `getNextReadyAndDiscard` primero busca la más prioritaria y luego, dentro de esa prioridad, la más corta. Una vez que la encuentra compara esta tarea con la tarea que se está ejecutando actualmente en el CPU. Si la tarea encontrada tiene mayor prioridad o es más corta entonces se le asigna al CPU la tarea encontrada, se la borra de `tasksReady` y la tarea que estaba siendo ejecutada actualmente en el CPU se guarda en su correspondiente vector del arreglo `tasksReady`. Si la tarea encontrada tiene menor prioridad o tiene la misma prioridad pero es más larga entonces se deja todo como estaba antes de llamar a la función `getNextReady` y se sigue ejecutando la tarea que estaba siendo ejecutada.

Lote y gráficos

Utilizaremos el siguiente lote y observaremos el gráfico resultante para certificar el comportamiento del nuevo scheduler.

```
TaskPriorizada 4 15
TaskPriorizada 3 20
@2:
TaskPriorizada 1 1
@3:
TaskPriorizada 2 2
@4:
TaskPriorizada 5 10
```

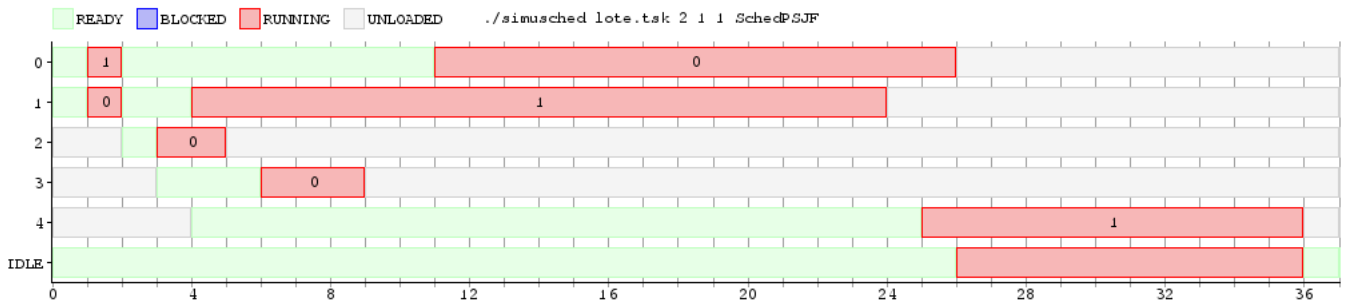

Figura 10: Gráfico del lote propuesto utilizando el scheduler *PSJF* para 2 núcleos y un cambio de contexto de 1 ciclo.

Se puede ver en la fig. 10 que el scheduler se comporta de forma deseada para el lote dado.

7. Ejercicio 7

Para este ejercicio propondremos el siguiente par de lotes y compararemos valores de *latencia*, *waiting time* y *throughput* entre los schedulers *SchedMistery*, *PSFJ* y el *Round-Robin*.

Lote 1 y Lote 1 bis

Para hacer una comparación justa entre los scheduler *PSJF*, *SchedMistery* y *Round-Robin*, proponemos el siguiente par de lotes, denominados `Lote 1` y `Lote 1 bis`, donde el `Lote 1` es un lote compuesto únicamente de tareas de tipo `TaskCPU` y el `Lote 1 bis` es un lote compuesto únicamente de tareas de tipo `TaskPriorizada` con la misma prioridad (1) en todas las tareas. Recordemos que `TaskPriorizada` y `TaskCPU` tienen la misma función.

Lote 1

```
TaskCPU 15
TaskCPU 20
@2:
TaskCPU 1
@3:
TaskCPU 2
@4:
TaskCPU 10
```

Lote 1 bis

```
TaskPriorizada 1 15
TaskPriorizada 1 20
@2:
TaskPriorizada 1 1
@3:
TaskPriorizada 1 2
@4:
TaskPriorizada 1 10
```

Scheduler *SchedMistery*

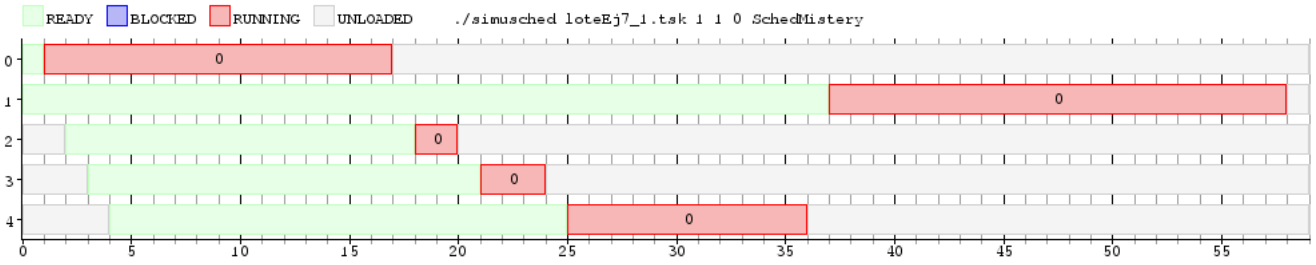

Figura 11: Gráfico del **Lote 1** utilizando el scheduler *SchedMistery* para 1 núcleo y un cambio de contexto de 1 ciclo.

Tarea	***Latencia***	***Waiting time***
Tarea 0	1	1
Tarea 1	37	37
Tarea 2	16	16
Tarea 3	18	18
Tarea 4	21	21
Promedio	18,6	18,6

Cuadro 6: Tabla de valores de *latencia* y *waiting time* de cada tarea y el promedio del **Lote 1** utilizando el scheduler *SchedMistery* para 1 núcleo y un cambio de contexto de 1 ciclo.

$$Throughput = \frac{5}{58} = 0,0862$$

Figura 12: Gráfico del **Lote 1** utilizando el scheduler *SchedMistery* para 2 núcleos y un cambio de contexto de 1 ciclo.

Tarea	*Latencia*	*Waiting time*
Tarea 0	1	1
Tarea 1	1	1
Tarea 2	16	16
Tarea 3	18	18
Tarea 4	19	19
Promedio	11	11

Cuadro 7: Tabla de valores de *latencia* y *waiting time* de cada tarea y el promedio del **Lote 1** utilizando el scheduler *SchedMistery* para 2 núcleos y un cambio de contexto de 1 ciclo.

$$Throughput = \frac{5}{34} = 0,1470$$

Viendo los cuadros 6 y 7 de las fig. 11 y 12 podemos ver que la *latencia* y el *waiting time* promedio da exactamente igual. Esto se debe a que el scheduler *SchedMistery* no utiliza nunca una política de *Preemptive*, es decir, una vez que elige una tarea, la misma no va a ser desalojada hasta que sea finalizada. Por lo tanto esto conlleva algunas desventajas. Por ejemplo, consideremos el siguiente lote:

```
1 TaskCPU 100000
2 @2:
3 TaskCPU 1
4 @3
5 TaskCPU 2
```

La *latencia* de las tareas 1 y 2 van a ser muy grandes y el *waiting time* de ellas también.

Por otro lado, en la política del *SchedMistery*, cuando muchas tareas se ponen en estado *ready* al mismo tiempo y el scheduler elige la más corta se garantiza que cada tarea va a tener una mejor *latencia* que si el scheduler eligiera al azar. Esto es porque al elegir la más corta, garantiza que la otra, que es más larga, va a esperar menos que si se eligieran al revés.

Scheduler *PSJF*

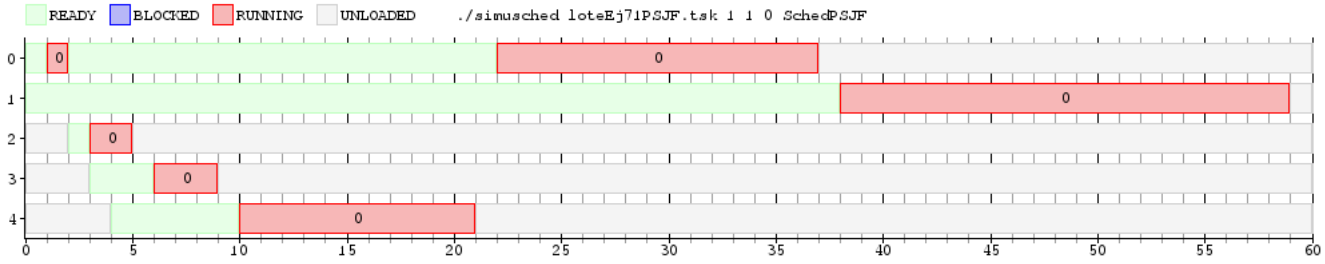

Figura 13: Gráfico del **Lote 1 bis** utilizando el scheduler *PSJF* para 1 núcleo y un cambio de contexto de 1 ciclo.

Tarea	*Latencia*	*Waiting time*
Tarea 0	1	12
Tarea 1	38	38
Tarea 2	1	1
Tarea 3	3	3
Tarea 4	6	6
Promedio	9,8	12

Cuadro 8: Tabla de valores de *latencia* y *waiting time* de cada tarea y el promedio del **Lote 1 bis** utilizando el scheduler *PSJF* para 1 núcleo y un cambio de contexto de 1 ciclo.

$$Throughput = \frac{5}{59} = 0,0847$$

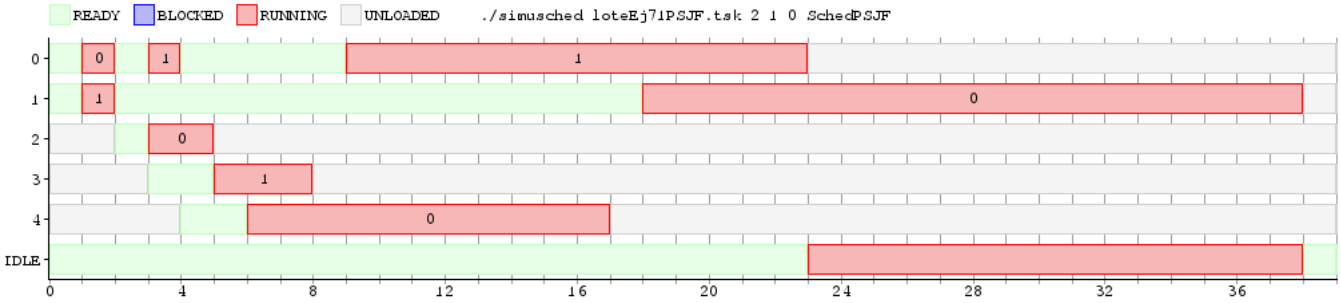

Figura 14: Gráfico del **Lote 1 bis** utilizando el scheduler *PSJF* para 2 núcleos y un cambio de contexto de 1 ciclo.

Tarea	*Latencia*	*Waiting time*
Tarea 0	1	7
Tarea 1	1	17
Tarea 2	1	1
Tarea 3	2	2
Tarea 4	2	2
Promedio	1,4	5,8

Cuadro 9: Tabla de valores de *latencia* y *waiting time* de cada tarea y el promedio del **Lote 1 bis** utilizando el scheduler *PSJF* para 2 núcleos y un cambio de contexto de 1 ciclo.

$$Throughput = \frac{5}{38} = 0,1315$$

En el **Lote 1 bis**, con un scheduler *PSJF* para 1 núcleo (cuadro 8), se puede observar bien el comportamiento *preemptive* del scheduler, ya que todas las tareas del lote tienen la misma prioridad y distintos usos del cpu . Las tareas mas cortas(como las tareas 2 y 3) van a tener una *latencia* y un *waiting time* más corto que las tareas con mayor uso de cpu (como la tarea 0 y 1). En Promedio, se obtiene una latencia mas baja que el *waiting time* gracias al comportamiento *preemptive* del scheduler.

Utilizando el mismo lote y utilizando el scheduler *PSJF* para dos núcleos, se puede ver una mejoría en la *latencia* y el *waiting time* son más bajos (cuadro 14). También se puede observar el comportamiento del scheduler ya que las tareas de mayor uso de CPU son desalojadas cuando llegan llegan tareas más cortas.

En comparación con la ejecución de un sólo núcleo, el *throughput* para dos núcleos es mucho menor ya que ejecuta la misma cantidad de tareas en un lapso de tiempo menor.

Scheduler *Round-Robin*

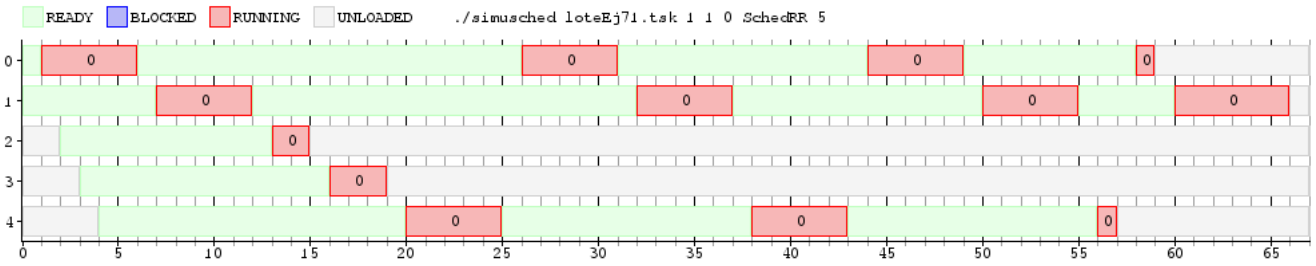

Figura 15: Gráfico del **Lote 1** utilizando el scheduler *Round-Robin* para 1 núcleo y un cambio de contexto de 1 ciclo.

Tarea	***Latencia***	***Waiting time***
Tarea 0	1	43
Tarea 1	7	45
Tarea 2	11	11
Tarea 3	13	13
Tarea 4	16	42
Promedio	9,6	30,8

Cuadro 10: Tabla de valores de *latencia* y *waiting time* de cada tarea y el promedio del **Lote 1** utilizando el scheduler *Round-Robin* para 1 núcleo y un cambio de contexto de 1 ciclo.

$$Throughput = \frac{5}{38} = 0,0757$$

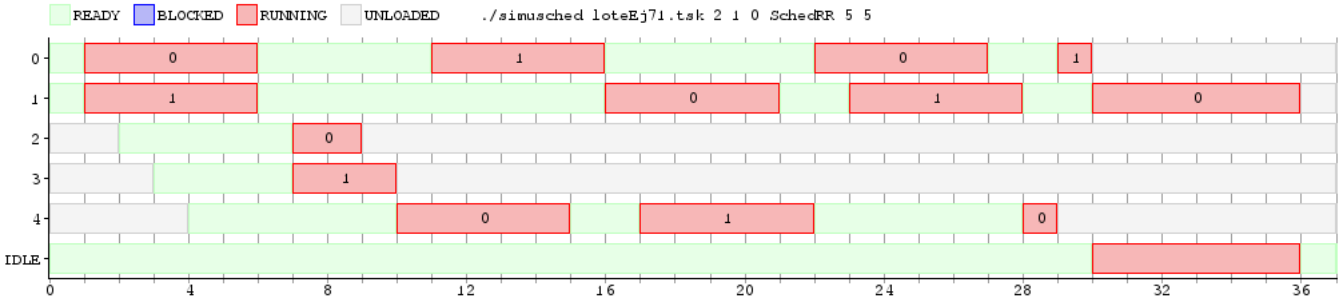

Figura 16: Gráfico del **Lote 1** utilizando el scheduler *Round-Robin* para 2 núcleos y un cambio de contexto de 1 ciclo.

Tarea	***Latencia***	***Waiting time***
Tarea 0	1	14
Tarea 1	1	15
Tarea 2	5	5
Tarea 3	4	4
Tarea 4	6	14
Promedio	3,4	10,4

Cuadro 11: Tabla de valores de *latencia* y *waiting time* de cada tarea y el promedio del **Lote 1** utilizando el scheduler *Round-Robin* para 2 núcleos y un cambio de contexto de 1 ciclo.

$$\textit{Throughput} = \frac{5}{38} = 0,1388$$

Viendo los cuadros 10 y 11 de las fig. 15 y 16 podemos ver que el *waiting time* es aproximadamente 3 veces la *latencia*, tanto con 1 núcleo como con 2 núcleos. Esto se debe a que la *latencia* en el scheduler *Round-Robin* suele ser baja dado que es un scheduler ”justo”(esto quiere decir que el scheduler reparte el procesador de forma justa dándole un tiempo similar a cada proceso). Además la *latencia* es baja ya que utiliza una política de *preemptive* por quantums.

Debido a esta política de *preemptive*, cada *n quantums*, la tarea pasa a ser desalojada y tiene que esperar $(n-1) \times quantums$ para volver a ejecutar, por lo tanto, el *waiting time* suele ser muy alto. Puede pasar que a una tarea le falte un *quantum* para terminar y pasa a ser desalojada y tiene que esperar a que se ejecuten todas las otras tareas. Esto es una desventaja, por ejemplo, veamos el siguiente lote de 20 tareas, donde la tarea 0 simula ser un videojuego.

```
TaskCPU 6
TaskCPU 1
TaskCPU 2
TaskCPU 1
TaskCPU 1
TaskCPU 2
TaskCPU 1
TaskCPU 1
TaskCPU 2
TaskCPU 1
TaskCPU 1
TaskCPU 2
TaskCPU 1
TaskCPU 1
TaskCPU 2
TaskCPU 1
@1:
TaskCPU 1
TaskCPU 2
TaskCPU 1
TaskCPU 1
```

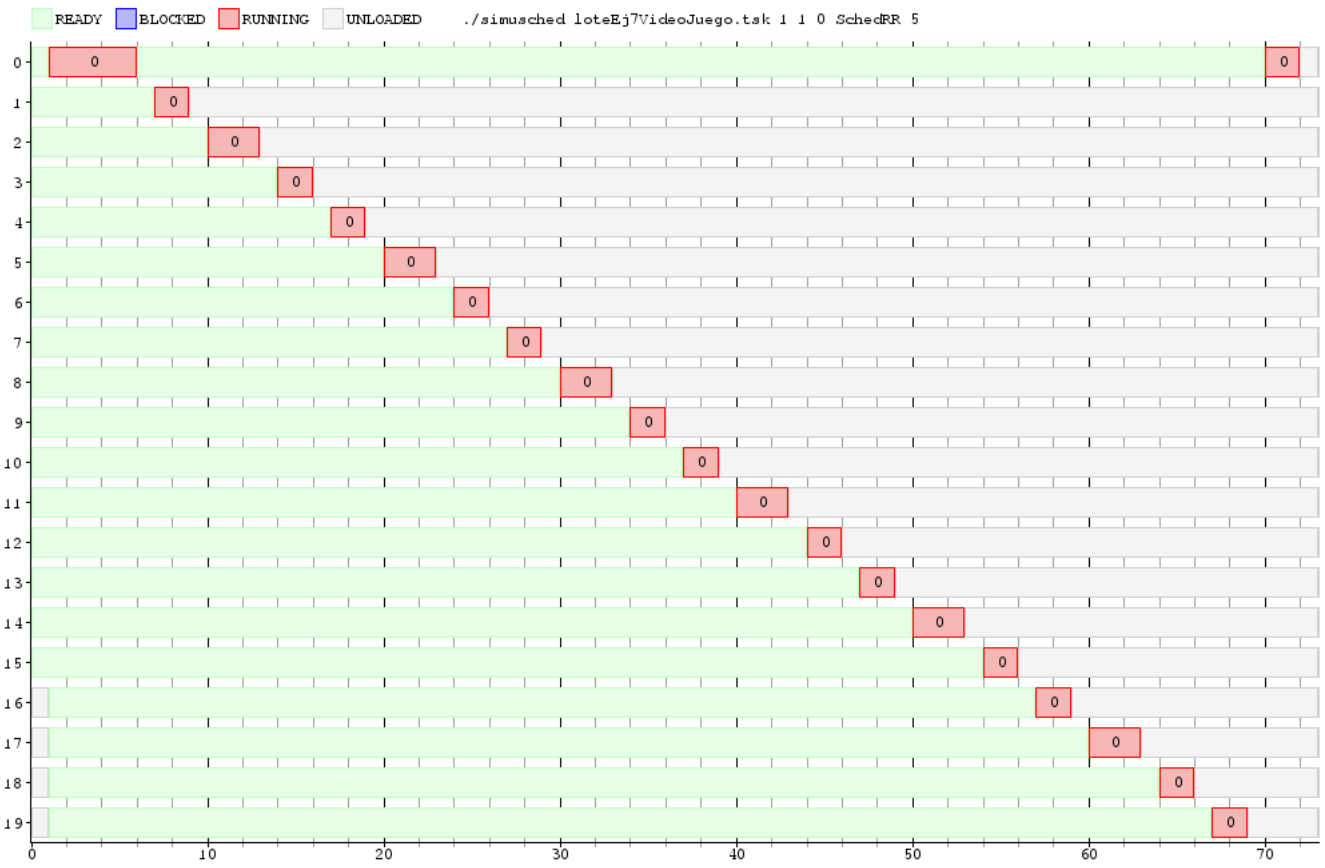

Figura 17: Gráfico del **Lote con Videojuego (tarea 0)** utilizando el scheduler *Round-Robin* para 1 núcleo y un cambio de contexto de 1 ciclo.

La *latencia* en promedio va a ser baja para todas las tareas, pero el *waiting time* del videojuego va a ser muy alto aunque le falte un sólo *quantum* para terminar.

Esto no sería lo deseado para un videojuego, donde el jugador no querría estar esperando para seguir jugando.

Como se ve en la fig. 17 nos dió el resultado esperado.

$$\textit{Latencia Videojuego} = 1$$

$$\textit{Waiting time Videojuego} = 96$$

Comparación de Schedulers

Scheduler	***Latencia***	***Waiting time***
SchedMistery	18,6	18,6
PSJF	9,8	12
Round-Robin	9,6	30,8

Cuadro 12: Tabla de valores de promedios de *latencia* y *waiting time* de cada scheduler de los **Lotes 1** y **bis** utilizando 1 núcleo y un cambio de contexto de 1 ciclo.

Scheduler	***Latencia***	***Waiting time***
SchedMistery	11	11
SchedPSJF	1,4	5,8
SchedRR	3,4	10,4

Cuadro 13: Tabla de valores de promedios de *latencia* y *waiting time* de cada scheduler de los **Lotes 1** y **bis** utilizando utilizando 2 núcleos y un cambio de contexto de 1 ciclo.

Como se ven en los cuadros 12 y 13, que comparan los promedios de *latencia* y *waiting time* entre los distintos schedulers, para el par de lotes dados, la mejor opción seria utilizar el scheduler *PSJF*.

Cuando hacemos comparaciones entre schedulers estaría mal decir que un tipo de scheduler es mejor o peor que otro. Esto se debe a que dependiendo cada situación particular, un tipo de scheduler o una mezcla de políticas de scheduler puede asociarse mejor o peor a un proyecto.

Veamos un ejemplo de esto:

Si tenemos una computadora de propósito general con un sistema operativo con entorno visual, veamos como afectaría aplicar las distintas políticas de scheduling.

Scheduler *Round-Robin*: Si aplicamos la política pura del scheduler *Round-Robin*, y tenemos muchas tareas en estado *ready*, se verá afectada la interacción del usuario con el sistema operativo, ya que un usuario no quiere que las tareas con entorno visual se reaccionen de forma lenta.

Scheduler *PSJF*: Aquí podríamos dividir las prioridades, dándole más prioridad a las tareas de entorno visual que van a ser lo que el usuario va a ver y a las tareas de fondo darle menos prioridad. Pero si una tarea tiene mayor prioridad de la que queremos ejecutar, esta nueva debera esperar demasiado para comenzar a ejecutarse. Esto se debe a que el scheduler únicamente desaloja una tarea si la misma es más prioritaria, o dentro de la misma prioridad, es más corta.

Scheduler *SchedMistery*: Aquí se verá muy afectada la interfaz visual, dado que si abrimos un programa y luego queremos abrir otro un segundo después, el programa nuevo no se abrirá hasta que termine de ejecutar el anterior. Por ejemplo, si abrimos el navegador *Chrome* y luego intentamos abrir el cliente de mensajería *Outlook*, este último no se abrirá hasta que el *Chrome* termine o lo cerremos.

Como vemos en este ejemplo, cada política pura no es 100 % efectiva. Es por esto que actualmente se utilizan muchas políticas combinadas para lograr el mejor propósito para el proyecto al que esté destinado el scheduler.

Quizás para un sistema operativo con entorno visual lo mejor sería darles prioridades altas a las tareas visuales y a la vez utilizar un scheduler de tipo *Round-Robin* dándole más *quantum* a las tareas que tienen más prioridad.