DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico: Scheduling

6 de septiembre de 2015 — Sistemas Operativos


| Integrante | LU | Correo electrónico |
|---|---|---|
| Arribas, Joaquín | 702/13 | joacoarribas@hotmail.com |
| Lebrero, Ignacio | 751/13 | ignaciolebrero@gmail.com |
| Vázquez, Jésica | 318/13 | jesis_93@hotmail.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

**Resumen**

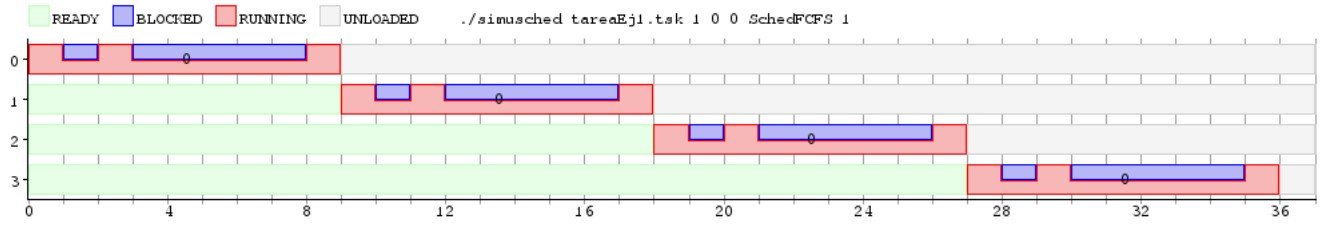
Con el crecimiento de los sistemas operativos y la capacidad de hardware para soportar varios procesos, surgen nuevos desafios a la hora de diseñar dichos sistemas. Uno de ellos es la organizacion de los procesos o *scheduling*, pero...¿hay una manera optima de hacerlo?, la respuesta es que depende cual sea la finalidad del sistema, Para esto existen muchos criterios bajo los cuales pueden ser organizados los procesos. En este trabajo se implementaron distintas simulaciones interactivas entre tareas. A su vez se implementaron distintas clases de scheduling para interactuar con las tareas creadas, y dichas interacciones se representaron de manera gráfica. Hare que no se hablar, i know that feling bro.

# 1. Ejercicio 1

El ejercicio consiste en implementar una tarea llamada **TaskConsola**, que simule una tarea que realiza llamadas bloqueantes. La tarea recibe por parámetro la cantidad de llamadas bloqueantes que debe realizar, y un intervalo que determina un máximo y un mínimo para la duración de cada una. Dicha duración es generada de manera pseudoaleatoria.

Para resolver el ejercicio creamos una función llamada **generate** que se encarga de realizar la simulación de la tarea. Genera una *semilla* utilizando la función **time** y luego, para cada llamada bloqueante, genera el tiempo usando la función **rand_r**. Para cada valor de la semilla se genera un valor pseudoaleatorio al cual se lo fuerza a caer en el intervalo pasado por parámetro, tomándole módulo la distancia entre el máximo y el mínimo, y luego sumándole el mínimo. Una vez calculado el tiempo, se hace la llamada al uso del dispositivo de I/O.

Ejemplo:




El gráfico muestra un lote de 4 tareas de tipo **TaskConsola**. El algoritmo de *scheduling* utilizado para representar la interacción entre las tareas es **First Come, First Served**. La cantidad de llamadas bloqueantes son 2 y el intervalo de tiempo para cada llamada es entre 2 y 6. Podemos observar como efectivamente la duración de cada llamada bloqueante pertenece a ese intervalo

## 2. Ejercicio 2

El ejercicio consiste en simular la situación que enfrenta nuestro querido amigo Rolando, el cual quiere correr un algoritmo a la vez que escucha música y consume drogas JAJAJAJAJAJAJAJAJA MUY BUENO. El algoritmo que corre hace uso intensivo del cpu por 100 ciclos, mientras que la música e internet realizan una cantidad determinada de llamadas bloqueantes. La música hace 20 e internet 25, cada una de duración variable entre 2 y 4 ciclos. La manera de generar la duración pseudoaleatoria de los ciclos de las llamadas bloqueantes fue la misma que la utilizada en el ejercicio previo, a través de la funcion **generate**. El algoritmo de *scheduling* utilizado para este ejercicio fue **First Come, First Served**.

Ejemplos: A contuacion analizaremos el rendimiento del señor rolando con una computadora de 1 y 2 cpus, costo de cambio de context 4 y scheduling $FCFS$.

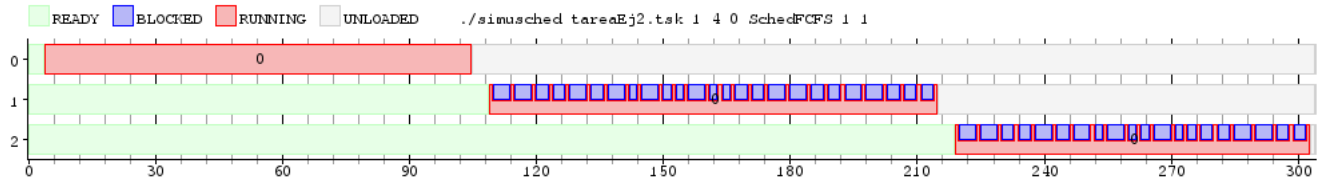



Figura 1: Simulación FCFS con cambio de contexto 4 en 1 cpu

Inserte metricas aqui ¡-

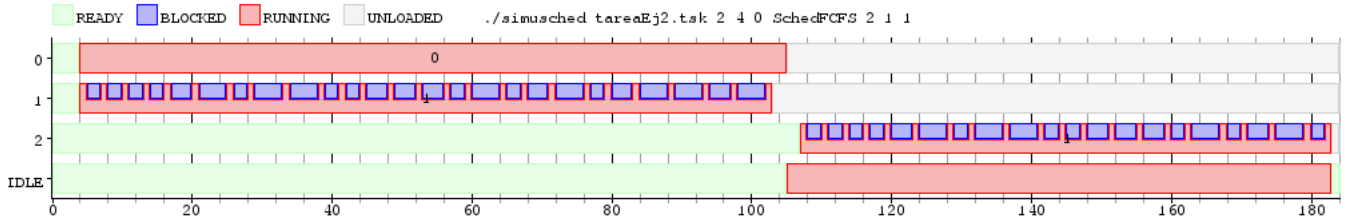



Figura 2: Simulación FCFS con cambio de contexto 4 en 2 cpus

## 3. Ejercicio 3

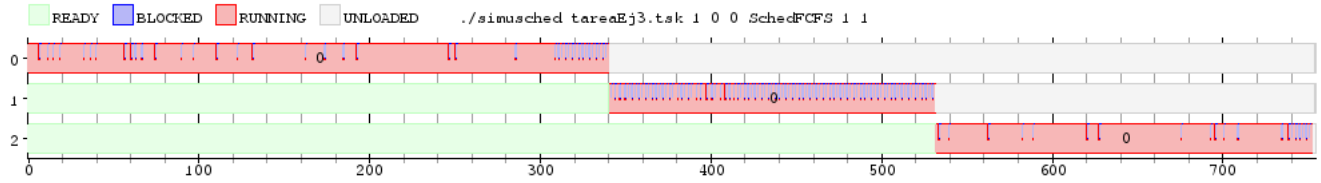



Figura 3: simulacion ejercicio 3

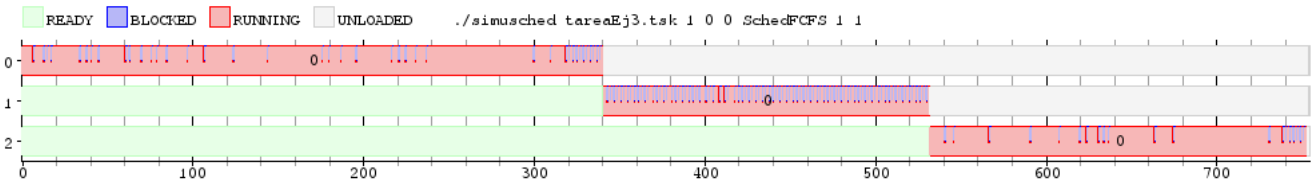



Figura 4: simulacion ejercicio 3

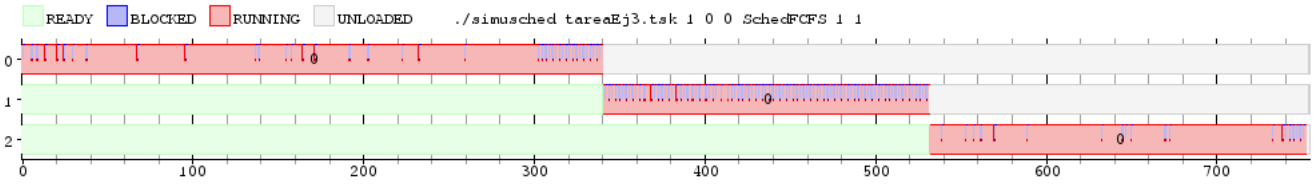



Figura 5: simulacion ejercicio 3

# 4. Ejercicio 4

En este ejercicio completamos la implentación del scheduler *Round-Robin*. Éste consiste en asignarle un quantum determinado a cada tarea e ir alternando el procesador entre las distintas tareas. Cada núcleo puede, o no, tener quantum distintos.

## 4.1. Elección de estructuras

Para la implementación del scheduler utilizamos una cola con las tareas que están en estado LISTAS, y almacenamos en un vector aquellas que están BLOQUEADAS. Además, cada núcleo conoce cuál es la tarea que está CORRIENDO, y cuántos ciclos lleva, con lo cual, dado que tiene un quantum determinado, calcula en qué momento debe desalojarla y darle el procesador a la siguiente tarea. Hay una única cola de tareas, en decir, se permite la migración de núcleos, ya que el scheduler desencola la primer tarea en estado LISTA y le asigna el núcleo que esté libre en ese momento sin importar en qué núcleo se ejecutó en ciclos anteriores.

# 5. Ejercicio 5

En esta sección evaluaremos el rendimiento del scheduler *Round-Robin* según el tiempo de latencia, el waiting-time (tiempo que el proceso está en estado READY), y el tiempo total de ejecución. Utilizaremos un lote de tareas con una de tipo **TaskCPU** de 50 ciclos, y dos **TaskConsola2** que hacen 5 llamadas bloqueantes, cada una de ellas de 3 ciclos de duración. Los cálculos de los cuadros están hechos sobre los datos de la simulación y la unidad de medida es *ciclo*.

## 5.1. Análisis de *Round-Robin* para distintos quantum

1. **Round-Robin con quantum 2**

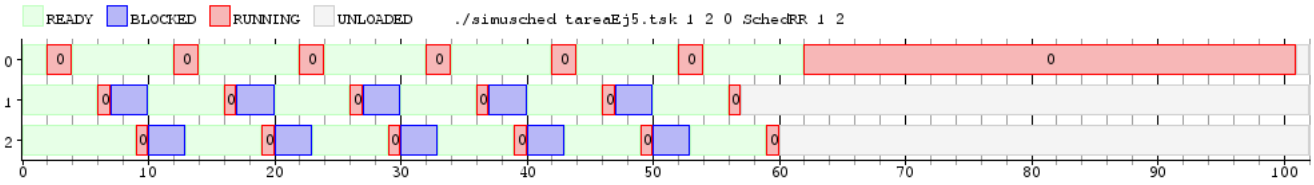



Figura 6: Simulación SchedRR quantum 2

| | Task0 | Task1 | Task2 |
|---|---|---|---|
| Latencia | 2 | 6 | 9 |
| Waiting Time | 50 | 36 | 39 |
| Tiempo de Ejecución | 101 | 57 | 60 |

Cuadro 1: cálculos con quantum 2

En este caso se observa que las 3 tareas tienen un tiempo de espera elevado con respecto al tiempo de ejecución total. La **Task0** espera durante el 50 % del tiempo, mientras que las **Task1** y **Task2** esperan aproximadamente 63 % y 65 % respectivamente. Por otro lado, la latencia de los procesos es baja con respecto a la cantidad de ciclos que le lleva terminar de ejecutarse.

2. **Round-Robin con quantum 10**

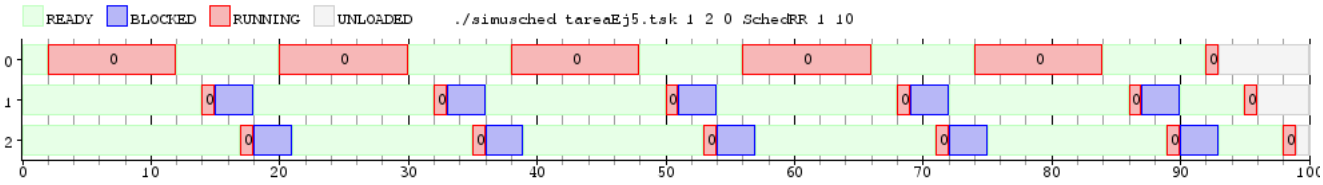



Figura 7: Simulación SchedRR quantum 10

Para el scheduler con quantum 10, se observa una mejora con respecto al tiempo de espera de la **Task 0**, disminuyendo el porcentaje a aproximadamente el 45,2 %, sin embargo, las otras dos tareas elevaron su tiempo de espera a alrededor del 78 %. Al igual que en el caso anterior, la latencia continúa siendo relativamente baja.

| | Task0 | Task1 | Task2 |
|---|---|---|---|
| Latencia | 2 | 14 | 17 |
| Waiting Time | 42 | 75 | 78 |
| Tiempo de Ejecución | 93 | 96 | 99 |

Cuadro 2: cálculos con quantum 10

3. **Round-Robin con quantum 50**

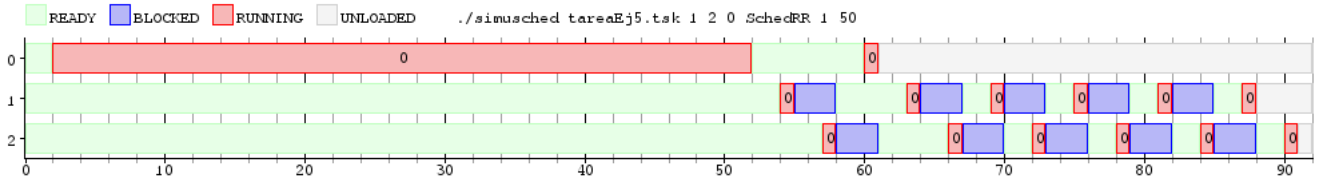



Figura 8: Simulación SchedRR quantum 50

| | Task0 | Task1 | Task2 |
|---|---|---|---|
| Latencia | 2 | 54 | 57 |
| Waiting Time | 10 | 67 | 70 |
| Tiempo de Ejecución | 61 | 88 | 91 |

Cuadro 3: cálculos con quantum 50

TODOS ESTOS PORCENTAJES ESTAN MAL CAPAAAAAAAAAA

En este caso, el porcentaje de tiempo de espera de la **Task 0** disminuye notablemente a un 16,4 %, y los tiempos de espera de las tareas restantes están alrededor de 76 %, porcentaje similar al caso del scheduler de quantum 10. Sin embargo, como en este caso se le asigna un quantum elevado a la primer tarea, el tiempo de latencia de las **Task 1** y **Task 2** ahora es de 54 y 57 ciclos respectivamente.

## 5.2. Conclusión

HARE LOCO

# 6. Ejercicio 6

En esta sección analizaremos las diferencias entre los schedulers *Round-Robin* y *First Come First Serve*. Para esto, simulamos el comportamiento del scheduler *FCFS* con el mismo lote de tareas del Ejercicio 5.

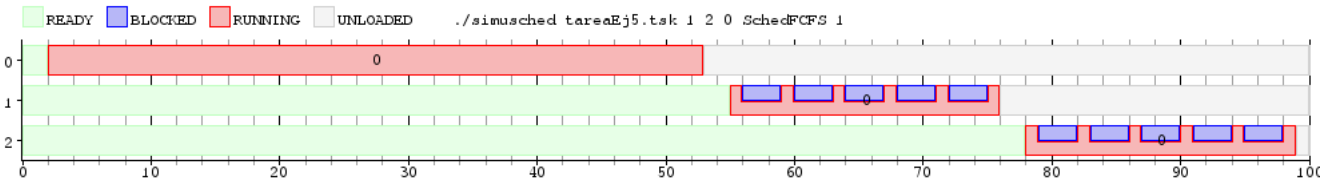



Figura 9: Simulación SchedFCFS

## 6.1. Comparación entre *SchedRR* y *SchedFCFS*

El comportamiento del *SchedRR* de quantum 50 para la **Task 0** es similar al del *SchedFCFS* dado que se ejecuta durante los 50 ciclos de duración (ver figura 3). El primer scheduler la desaloja en el ciclo 51 (aumentando su waiting time dado que solo le faltaba ejecutar el EXIT), mientras que el otro la desaloja una vez que realizó el EXIT. Como puede observarse en las dos figuras, el tiempo de latencia es similar al igual que el tiempo total de ejecución. Sin embargo, el *SchedRR* aprovecha el hecho de que una tarea esté bloqueada para darle procesador a la siguiente, como se ve en el ciclo 55 de la figura 3: la **Task 1** tenía el procesador, y, apenas se bloquea, el scheduler la desaloja y pone a correr la **Task 2** que comienza a ejecutar en el ciclo 57 dado que el costo de cambio de contexto es de 2 ciclos.

En el *SchedRR* de quantum 10, la **Task 0** tiene un tiempo de ejecución mayor que en el caso anterior, dado que, si no hay bloqueos, el scheduler le da el procesador a la siguiente tarea cada 10 ciclos. En este caso, esto no es ventajoso, dado que las **Task 1** y **Task 2** apenas empiezan a ejecutar, se bloquean, entonces la tarea 10 paga el costo de cambio de contexto al momento de volver a tener el procesador, pero las otras dos tareas ejecutaron solamente 1 ciclo (por ejemplo, en el ciclo 12 de la figura 2, la tarea 0 es desalojada, luego de 2 ciclos de costo de cambio de contexto, ejecuta la tarea 1 y se bloquea, entonces el scheduler le pasa el procesador a la tarea 2, que hace lo mismo que la 1, luego éste la desaloja, y la tarea 0 paga el costo de cambio de contexto para volver a ejecutar). Pablito clavo un clavito apesta esto

Por último, el *SchedRR* de quantum 2, YA NO SE QUE PONER ACA.

## 6.2. Conclusión

El *SchedRR* es más justo con respecto a la asignación de CPU. Aunque eventualmente el *SchedFCFS* le da procesador a todas las tareas, una con muy pocos ciclos de ejecución podría esperar que termine otra que tiene una cantidad significativamente mayor de ejecución. En el primer scheduler, probablemente la tarea de tiempo de ejecución menor pueda terminar antes, sin tener que esperar que todas las demás hayan terminado de usar el procesador. Esto también se ve reflejado en la latencia, ya que en FCFS un proceso podría estar esperando una cantidad indefinida de tiempo (dependiendo del momento en el que se inicio, cuántos procesos hay en la cola y cuánto tiempo van a tardar) mientras que en RR va a tener una latencia razonable dependiendo del quantum del scheduler. Es por esto que el *Turnaround* de cada proceso va a ser menor, ya que como se va alternando el uso de procesador, no van a tener que sumar todo el tiempo que estuvieron esperándolo.

# 7. Ejercicio 7

En esta sección experimentamos con el *SchedMistery*, a continuación ejemplos de su comportamiento:

1. **Pasar como parámentro al scheduler un** 0

   Esta simulación está hecha con el mismo lote de tareas del Ejercicio 5 (una **TaskCPU** de 30 ciclos y dos de 30). Los valores pasados como parámetro fueron 5 4 0 4.

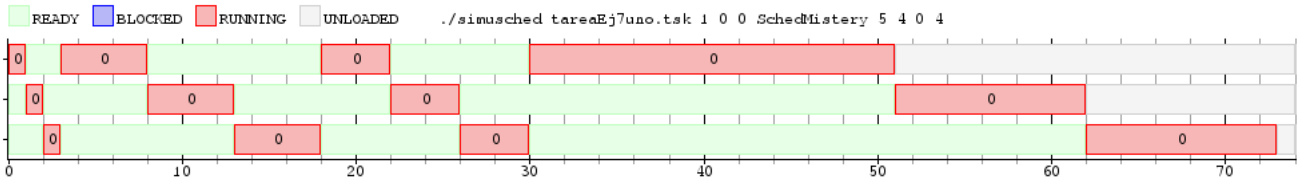



Figura 10: Simulación SchedNoMistery con un 0 como parámetro

   En este caso, observamos que todas las tareas corren 1 ciclo, 5 ciclos, 4 ciclos y cuando el scheduler toma el valor 0 ejecuta los procesos hasta el EXIT sin importar los parámetros que fueron pasados después.

2. tu vieja

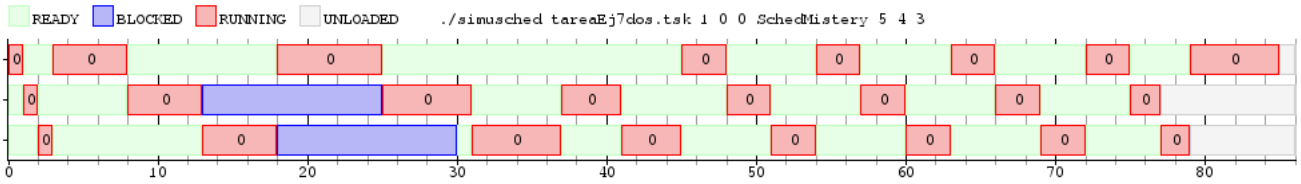



Figura 11: Simulación SchedNoMistery con bloqueos

   En esta simulación, corre la **Task0** durante 5 ciclos

## 7.1. Ejercicio 8

En este ejercicio implementamos un shceduler *RoudRobin* con la propiedad de que no permite migracion de procesos entre cpus, a continuacion se analizara su performance en comparacion con *RoundRobin*(*hood*) implementado en el ejercicio 4.

Veamos cuando esto no es conveniente, supongamos que tenemos un varios cpus y uno de ello posee quantum grande, entonces si alguna tarea que no bloquee cae sobre ese cpu, las tareas que corran sobre el y contengan bloqueos se veran afectadas, ya que no usaran todo su quantum y luego de haber terminado el periodo de bloqueo deberan esperar probablemente un gran periodo de tiempo hasta poder volver a correr, veamos un ejemplo:

Para este ejemplo se utilizaron *TaskRolando* y *TaskConsola* para simular las llamadas a CPU intensivas y los bloqueos constantes sobre el primer CPU y sobre el segundo simplemente se corrio una tarea rolando.

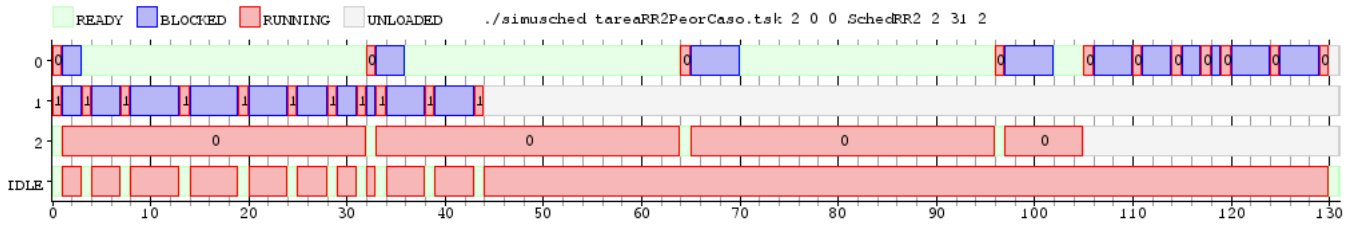



Figura 12: Simulacion schedRR2 para peor caso en Ejercicio8

A simple vista se puede sobservar que el cpu 1 pasa la mayor parte de su tiempo en idle esperando que la tarea que corre se desbloquee.

A continuacion veremos como hubiera corrido si hubieramos usado el scheduler del ejercicio 4

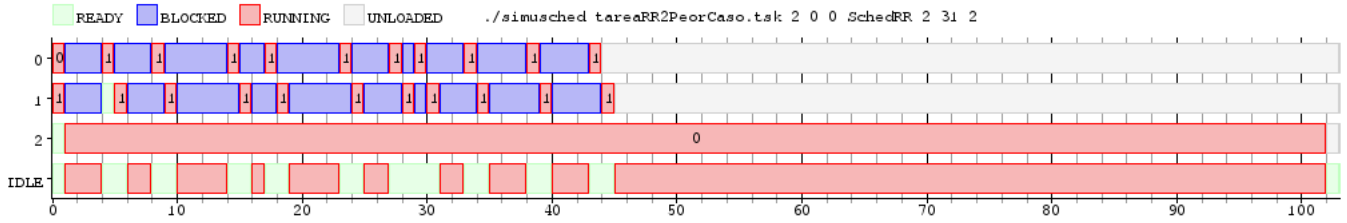



Figura 13: Simulacion schedRR para peor caso en Ejercicio8

Finalmente, veremos como funcionaria bajo *FirstComeFirstServe*

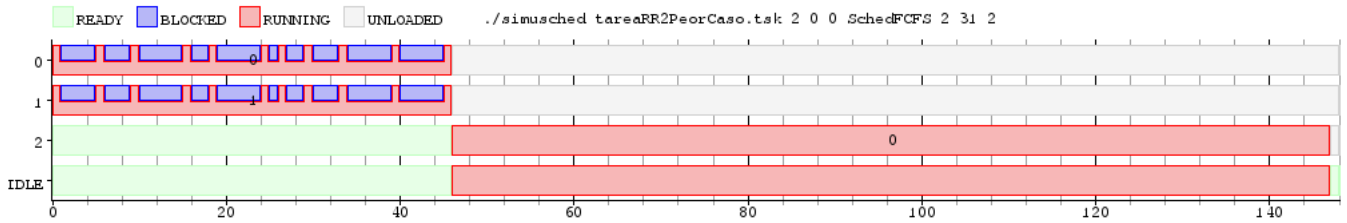



Figura 14: Simulacion *FistComeFirstServe* para peor caso en Ejercicio8

Comparando los resultados, se puede observar que utilizando *FCFS* cuesta casi lo mismo que no permitir que los procesos cambien de CPU.

Ahora veamos el caso donde esto si sea conveniente, supongamos que tenemos un costo elevado para cambiar el proceso de nucleo, tenemos muchos procesos y cada CPU tiene poco quantum. En este caso se espera que los procesos sean intercambiados entre si muchas, de esta manera no esperaran demasiado y podran ir ejecutando de a poco. veamos un ejemplo:

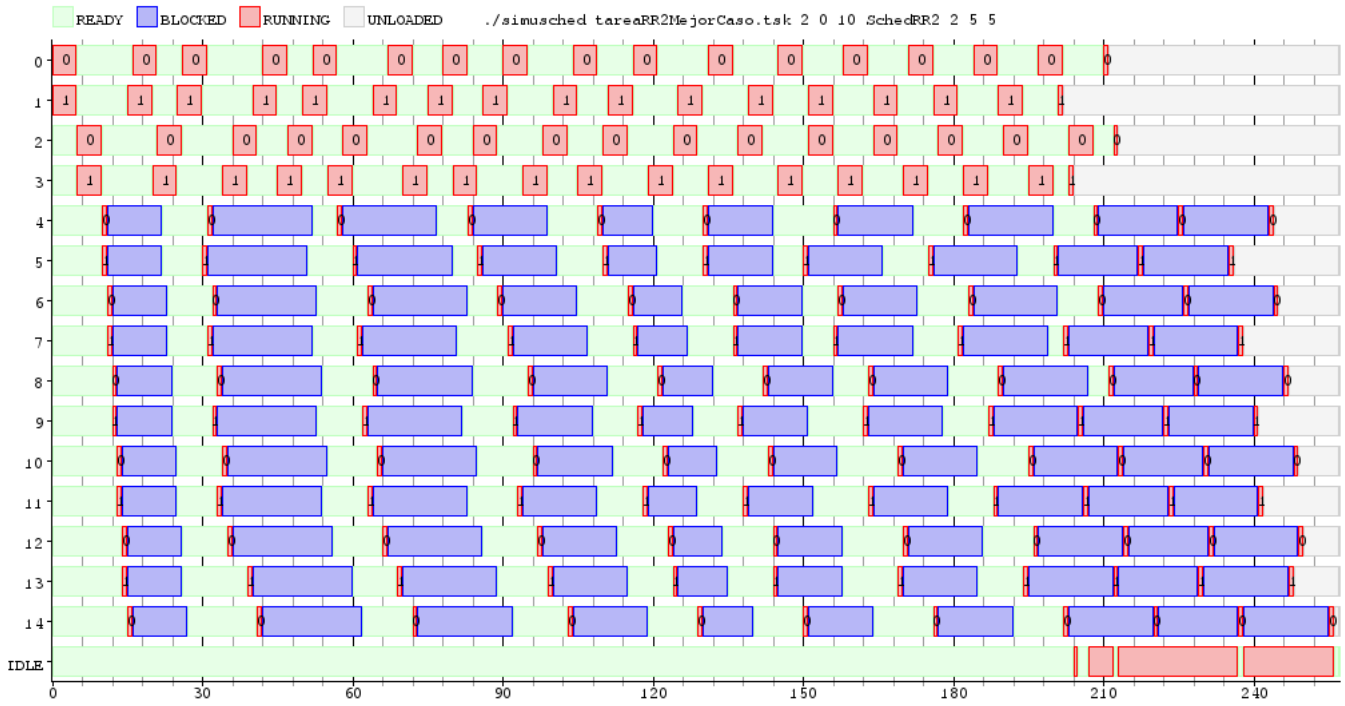



Figura 15: Simulacion schedRR2 para mejor caso en Ejercicio8

Como podemos observar, los procesos se reparten en los dos cpus y corren intermitentemente, como nunca cambian de procesador no pagan el costo de hacerlo. veamos ahora que pasaria en un entorno *roundrobin* comun.

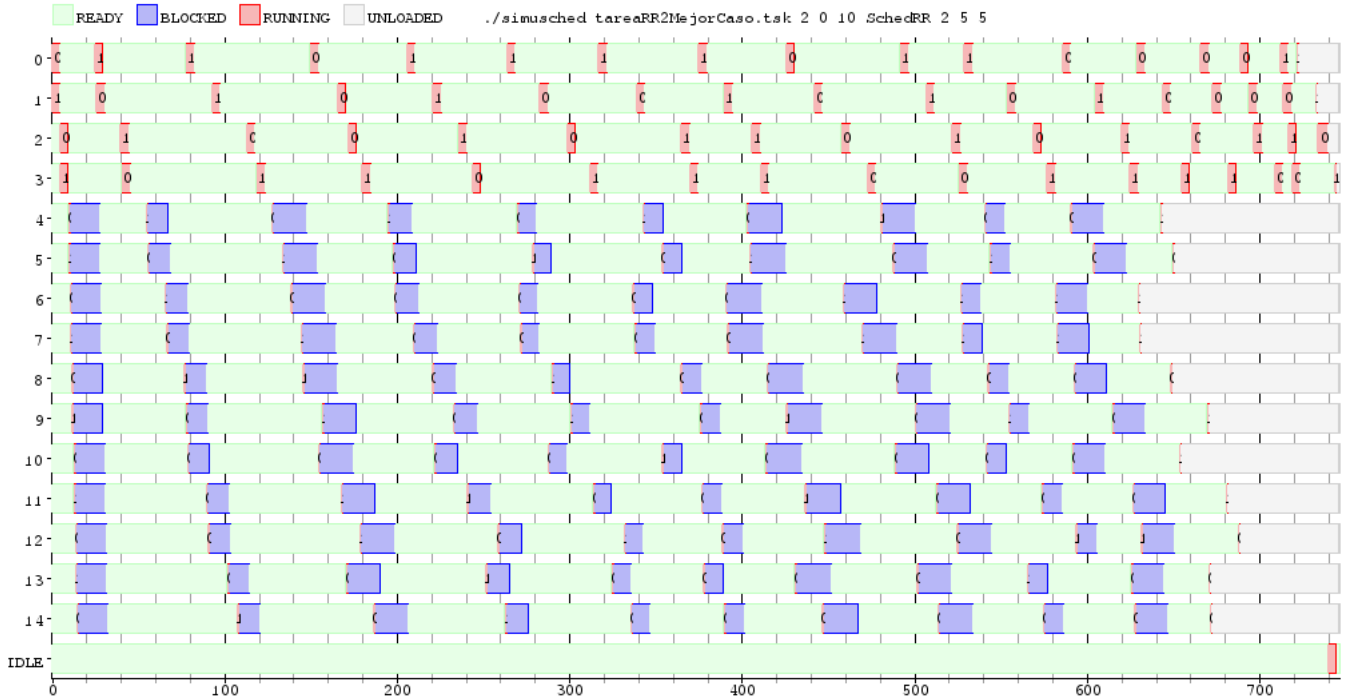



Figura 16: Simulacion schedRR para mejor caso en Ejercicio8

En este entorno los procesos intercambian cpus constantemente, con esto pagan siempre el

costo de hacerlo por lo que demora enormemente que terminen de correr.

Ahora veamos como correria en un contexto *FirstComeFirstServe*

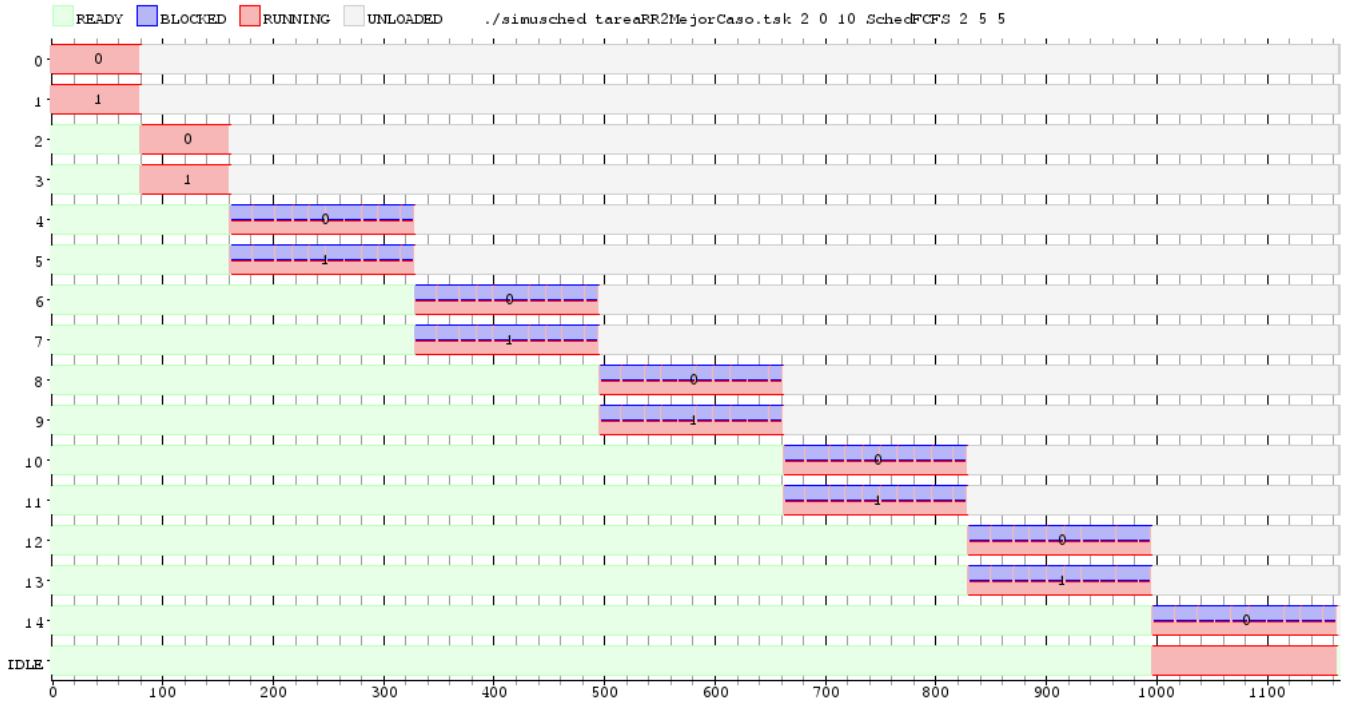



Figura 17: Simulacion *First Come First Serve* para mejor caso en Ejercicio8

En este caso no se paga nada adicional, ya que los procesos empiezan cuando se les da el procesador y no iniciara el proximo hasta que este no termine, por lo que nunca pagara el costo de intercambio de cpu. Por otro lado el costo total que tomara procesar todo sera muchisimo mayor, ya que cuando algun proceso se bloquee no habra posibilidad de poner otro a correr en el medio haciendo que el total de corrida de los procesos sea mucho mayor.