

Sistemas Operativos

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico Número 1

Scheduling

Integrante	LU	Correo electrónico
Ciruelos Rodríguez, Gonzalo	063/14	gonzalo.ciruelos@gmail.com
Thibeault, Gabriel	114/13	gabriel.eric.thibeault@gmail.com

Índice

1. Ejercicio 1	3
2. Ejercicio 2	3
3. Ejercicio 3	4
4. Ejercicio 4	5
5. Ejercicio 5	5
6. Ejercicio 6	6
7. Ejercicio 7	6
8. Ejercicio 8	7
8.1. Escenario 1	8
8.2. Escenario 2	9

1. Ejercicio 1

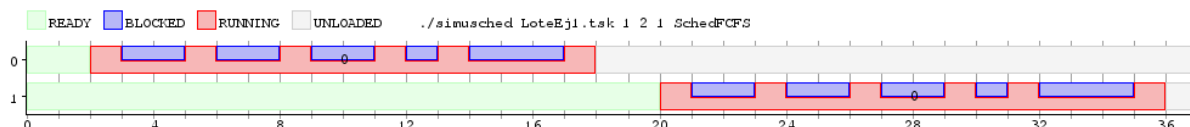
Nuestra implementación de la TaskConsola consiste en un ciclo que realiza n (el primer parámetro de la tarea) interrupciones. Para determinar la cantidad de tiempo que debe durar cada interrupción empleamos la función `std::rand()` de la librería estándar. Sin embargo, es necesario normalizar el resultado de ésta, ya que devuelve un valor entre 0 y `RAND_MAX` (un macro definido en `stdlib.h`, dependiente de la implementación) y necesitamos que esté entre $bmin$ y $bmax$ (los otros dos parámetros de la tarea).

Para lograr esto, dividimos el valor aleatorio inicial por $RAND_MAX - 0$, para obtener un número perteneciente al intervalo $[0, 1]$. Luego, lo multiplicamos por $(bmax - bmin) + 1$, obteniendo un resultado perteneciente al conjunto $[0, bmax - bmin + 1]$. Finalmente sumamos $bmin$ y obtenemos un valor perteneciente al conjunto $[bmin, bmax + 1]$. Si bien esto puede parecer incorrecto, el cast de un `double` a un `int` en C++ toma la parte entera del `double`, por lo que este conjunto es el deseado.

El único caso donde el resultado puede no llegar a ser el deseado es donde el valor es exactamente $bmax + 1$. Si bien esto no puede pasar en una variable continua (como sería teóricamente la uniforme que estamos manejando), al tratar con una representación de máquina de los reales (como son los `doubles`) esto es posible (y sucede cuando `rand()` devuelve exactamente `RAND_MAX`). Para evitar este error, si el resultado final es exactamente $bmax + 1$, le asignamos $bmax$.

En la figura 1 se puede ver un ejemplo de un lote de tareas con dos Task Consolas (ambas con parámetros $n = 5$, $bmin = 1$ y $bmax = 3$).

Figura 1: Ejemplo de un lote de tareas con Task Consola



2. Ejercicio 2

En las figuras 2 y 3 se pueden ver los gráficos de Gantt para el lote de tareas de este ejercicio, para 1 y 2 cores, respectivamente.

La latencia para un solo core es de 4 ciclos de CPU para la taskCPU, 109 ciclos para una de las tasksConsola y 190 ciclos para la otra. Para dos cores es de 4 ciclos para la taskCPU y una de las tasksConsola y de 85 para la otra.

Con los resultados de la figura 2 se puede ver claramente el problema que puede surgir al utilizar un scheduler First Come First Serve: el usuario en este tipo de casos esperaría poner a correr su algoritmo, y pasar el tiempo hasta que éste termine de ejecutarse utilizando otras aplicaciones. Sin embargo, al correr este lote de tareas en una computadora con un solo core se debe esperar a que termine de correr la primer tarea para poder utilizar otras

aplicaciones.

Si bien con otro tipo de scheduler (por ejemplo Round Robin) el algoritmo tardaría más tiempo, el usuario podría utilizar otras aplicaciones mientras tanto para hacer pasar el tiempo.

Al emplear dos cores para correr el lote de tareas se alivia el problema detallado previamente, ya que al menos se puede ejecutar una de las aplicaciones deseadas mientras corre el algoritmo CPU-intensivo en el otro core, como se ve en la figura 3. Sin embargo, el problema no se elimina completamente debido a que aún se debe esperar a que la primera aplicación termine de ejecutar para poder correr la segunda.

La única forma de eliminar completamente este problema para un scheduler FCFS es tener a disposición más cores que tareas a ejecutar, pero sólo en raras circunstancias es posible satisfacer este requerimiento.

Figura 2: Lote de tareas de Rolando para 1 core

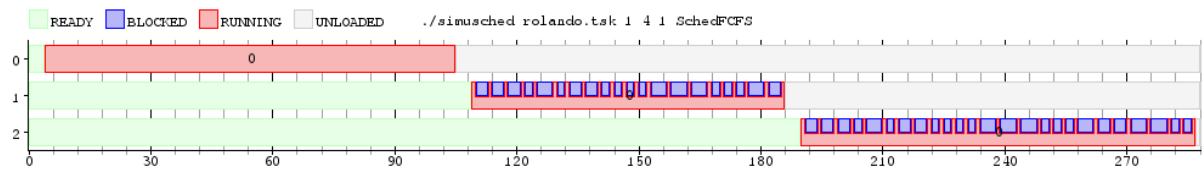
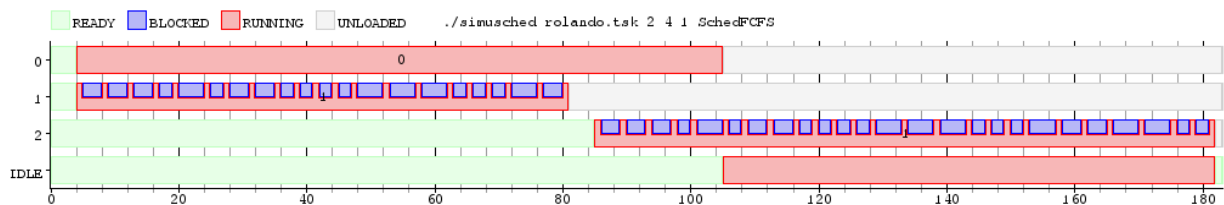


Figura 3: Lote de tareas de Rolando para 2 cores

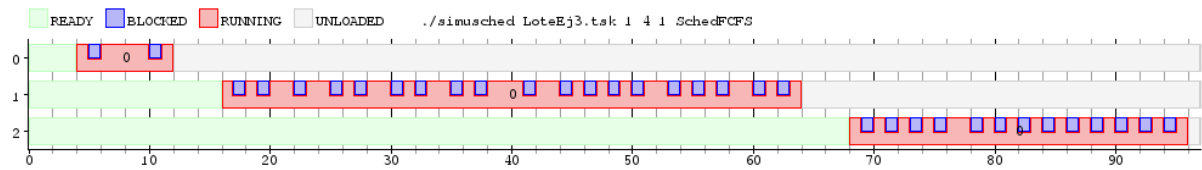


3. Ejercicio 3

Nuestra implementación de Task Batch consiste en crear un vector de bools de tamaño *total_cpu* (el primer parámetro de la tarea), que se llena con *cant_bloqueos* valores en *true* y el resto en *false*. Luego, utilizamos el algoritmo de Fisher-Yates para hacer un shuffle del vector. Finalmente, se lo recorre, y por cada posición, si el valor es *true* se hace una interrupción de IO, y si es *false* se hace uso de la CPU.

En la figura 4 se puede ver un lote de tareas con 3 TaskBatch, con parámetros 5 y 2; 28 y 19; 14 y 13.

Figura 4: Ejemplo de un lote de tareas con Task Batch



4. Ejercicio 4

5. Ejercicio 5

Figura 5: Lote de tareas ejecutando con Round Robin con quantum 2

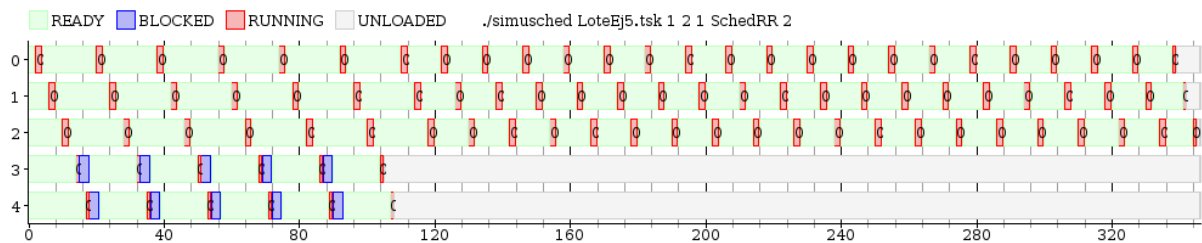


Figura 6: Lote de tareas ejecutando con Round Robin con quantum 10

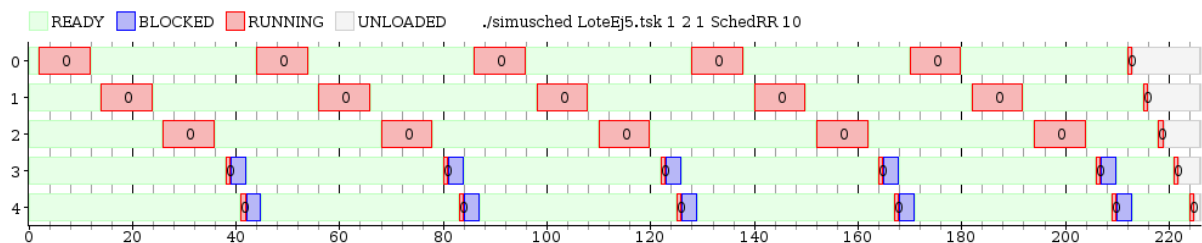
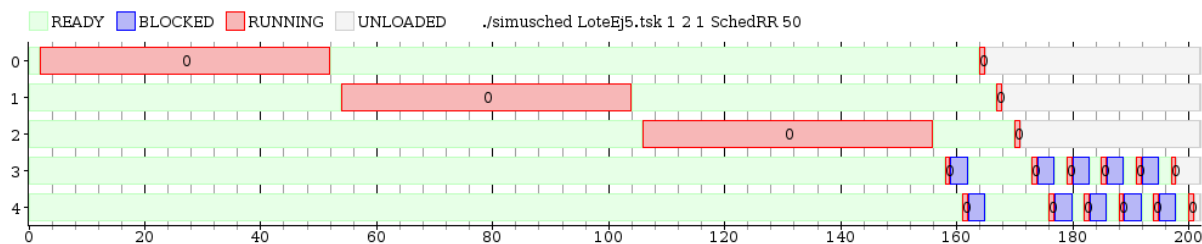


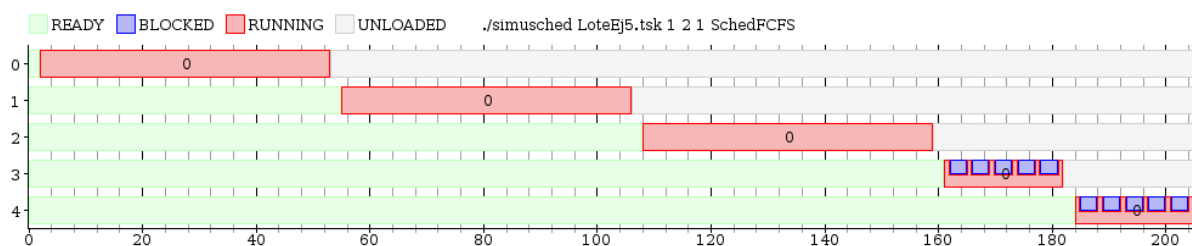
Figura 7: Lote de tareas ejecutando con Round Robin con quantum 50



Quantum	2	10	50
Latencia			
Waiting Time			
Tiempo Total			

6. Ejercicio 6

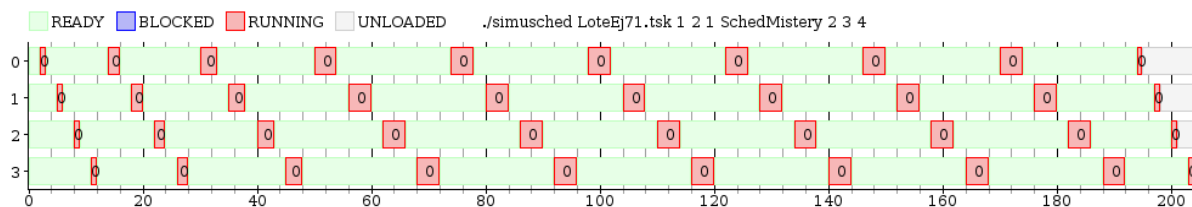
Figura 8: Lote de tareas ejecutando con FCFS



7. Ejercicio 7

Primero, lo que hicimos fue testear cuando se le pasaban muchos parámetros.

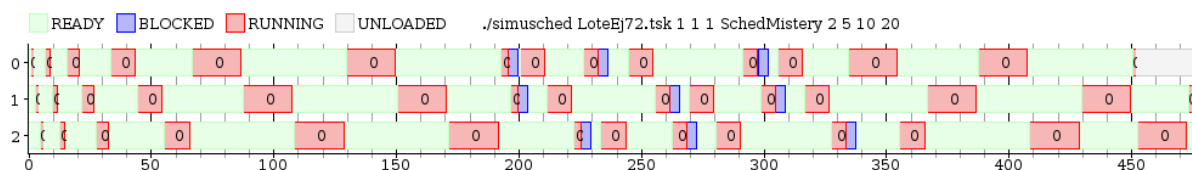
Figura 9: Ejemplo de un lote de tareas con Task Consola



Como se vé en la imagen, la cantidad de parámetros determina cuanto va a ser el quantum que se le asigna a cada tarea, progresivamente. Además gracias a este test, pudimos ver que hay preemption, y que el método probablemente sea similar a Round Robin.

Luego, lo que hicimos fue ver que pasa si una tarea se bloquea.

Figura 10: Ejemplo de un lote de tareas con Task Consola



Gracias a esta imagen pudimos confirmar que hay una idea de prioridades detrás de este scheduler. Esto se debe a que cuando una tarea se desbloquea, tiene *más prioridad*, dado que por ejemplo, la tarea 0, después de desbloquearse, se ejecuta antes que la tarea 2, que era la que a priori le tocaba, si se tratara de un Round Robin común y corriente.

Además, este test nos permitió ver que cuando una tarea se desbloquea va a la cola que le sigue en prioridad (de más prioridad).

Entonces, de esta manera confirmamos que se trata de un scheduling de prioridad, en el que hay n colas (si nos pasaron $n - 1$ parámetros), donde cada cola tiene un quantum igual al parámetro que corresponde (o 1 si es la primera). Y lo que hace el algoritmo es ir por la lista de colas buscando alguna cola no vacía, y si encuentra alguna no vacía corre la próxima tarea que corresponda.

Además, cuando una tarea es desalojada, pasa a estar en la cola siguiente (de menor prioridad) que la que estaba antes, y si estaba en la de menos prioridad se queda en esa.

En cuanto a la implementación, tenemos un vector de colas, y un vector de quantums (cuanto es el quantum de cada cola de tareas). Además, tenemos un entero que nos indica cuanto tiempo le queda a la tarea actual y de que cola salió.

Entonces, cuando cargamos una tarea la ponemos en la cola mas prioritaria. Cuando una tarea se desbloquea, la ponemos en la cola anterior (de más prioridad). Cuando una tarea debe ser desalojada porque se acabó el tiempo que se le asignó, se recorre el vector de colas buscando la primera no vacía y en caso de que la encuentre, popea un pid y lo pone a correr. En caso de que no encuentre, continúa corriendo la tarea actual.

8. Ejercicio 8

Para implementar este scheduler, debemos mantener n cantidad de colas, donde n es la cantidad de cores del procesador. Además, como cuando entra un proceso nuevo debemos asignarlo al procesador con menos procesos totales, nos conviene tener un vector que nos dice cuantos procesos totales tiene cada core. Como siempre, tenemos los quantums de cada procesador, y el tiempo restante que tiene cada tarea de cada core.

Finalmente, tenemos un diccionario, cuyas claves son tareas y sus significados son cores, que nos dice para cada proceso bloqueado, a que core le corresponde. Necesitamos esto para saber, cuando un proceso se desbloquea, a que core asignárselo.

Entonces, cuando entra un nuevo proceso, buscamos cual es el core con menos procesos actuales, y lo asignamos ahí. Cuando un proceso termina, lo que hacemos es fijarnos en la cola de cada core cual es el proceso que sigue.

Cuando un proceso se bloquea, lo definimos en el diccionario nombrado anteriormente; y cuando se desbloquea, buscamos en el diccionario a que core tiene que ir y borramos su entrada.

8.1. Escenario 1

La versión de Round Robin que no migra núcleos es beneficiosa en conjuntos de tareas que usan intensivamente el CPU y la memoria durante mucho tiempo. Como la memoria se usa intensivamente, migrar cores es costoso porque se pierde todo lo que estaba cacheado.

El lote que diseñamos consiste en 3 tareas, cada una empieza un segundo despues de la otra, y cada una usa el procesador por 30 unidades de tiempo.

```

1 TaskCPU 30
2 @1:
3 TaskCPU 30
4 @2:
5 TaskCPU 30

```

Figura 11: Ejemplo de un lote de tareas que solo usa CPU en Round Robin 1

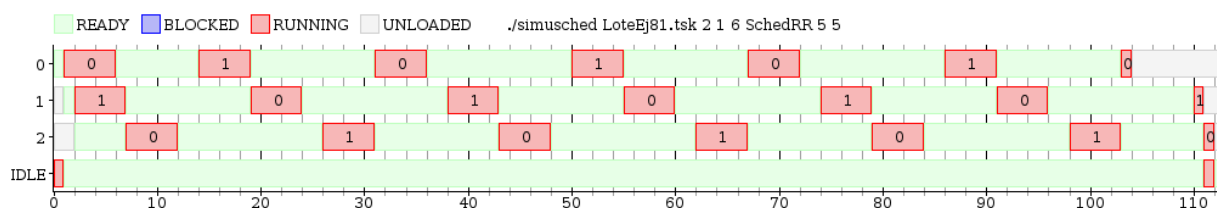
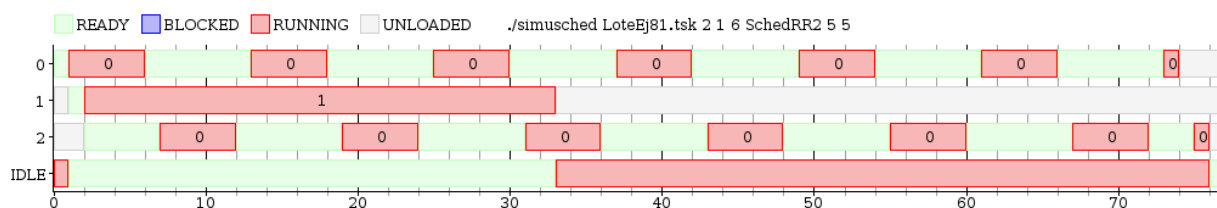


Figura 12: Ejemplo de un lote de tareas que solo usa CPU en Round Robin 2



Las métricas que analizaremos serán la eficiencia (tiempo en el que el CPU está corriendo las tareas sobre tiempo total) y el throughput.

Como se ve en la figura 11, la eficiencia de el algoritmo de scheduling que migra núcleos no es tan alta, dado que la migración de núcleos ocurre muy seguido y se debe pagar un costo muy alto cada vez. Por esta misma razón, el throughput de este algoritmo es también bajo.

Por otro lado, como se ve en la figura 12, la eficiencia de el algoritmo de scheduling que no migra núcleos (si no fuera por el idle del final, que se solucionaría simplemente poniendo igual cantidad de tareas en cada núcleo) es mucho más alta. Esto se debe a que al

no migrar núcleos, se aprovecha el cache del core y no se paga el costo de perderlo. Por esta razón, el throughput del algoritmo es más alto que antes, terminando las tareas en casi la mitad del tiempo que el anterior algoritmo.

8.2. Escenario 2

Ahora veamos que pasa si hay una tarea que se bloquea por mucho tiempo, una que lo usa por muy poco tiempo y otras 2 que requieren usar CPU por un poco más de tiempo.

```

1 TaskIO 1 70
2 TaskCPU 50
3 @1:
4 TaskCPU 5
5 TaskCPU 50

```

Figura 13: Ejemplo de un lote de tareas que se bloquean y usan CPU en Round Robin 1

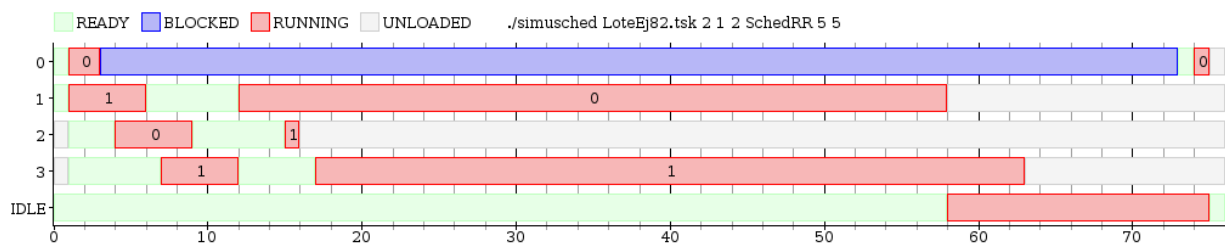
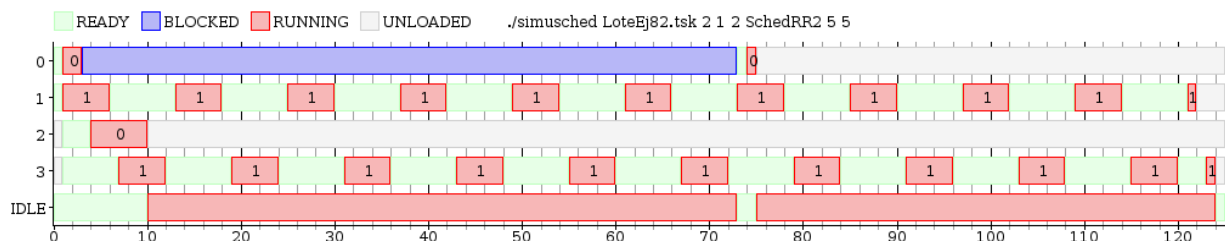


Figura 14: Ejemplo de un lote de tareas que se bloquean y usan CPU en Round Robin 2



Como se ve en la figura 13, el rendimiento y el throughput de esta versión es altísimo. Esto se debe a que, cuando se bloquea un proceso, el algoritmo puede correr en paralelo las 2 tareas demandantes de CPU en paralelo totalmente.

Sin embargo, en la versión del algoritmo que no permite migración entre procesadores, en la figura 14 todo cambia. Al no poder migrar tareas entre núcleos, cuando la tarea 2 termina, y la tarea 0 está bloqueada, el procesador 1 tiene una carga muy alta, mientras que el procesador 0 esta totalmente ocioso. Por esta razón, la eficiencia es mucho más baja que antes, y en consecuencia, el throughput es mucho más bajo.