



**DEPARTAMENTO
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Scheduling

2 de Mayo de 2016

Sistemas Operativos

Integrante	LU	Correo electrónico
Costa, Manuel José Joaquín	035/14	manucos94@gmail.com
Coy, Camila		
Ginsberg, Mario Ezequiel	145/14	ezequielginsberg@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	3
2. Ejercicio 2	4
3. Ejercicio 3	6
4. Ejercicio 4	7
5. Ejercicio 5	8
6. Ejercicio 6	10
7. Ejercicio 7	11
8. Ejercicio 8	12

1. Ejercicio 1

Para crear la tarea pedida, creamos un nuevo método en el archivo `task.cpp` llamado `TaskConsola`. El código que implementa la solución es sumamente simple.

La tarea itera sobre la cantidad de llamadas bloqueantes que el usuario pasó por parámetro, generando por cada iteración un número pseudoaleatorio, `nro_random`, y realizando la respectiva llamada de *I/O* (cuya duración es `nro_random`).

Notar que para aumentar el grado de aleatoriedad inicializamos el valor de la semilla a partir del tiempo del sistema, en lugar de un valor fijo, cada vez que se llama a `TaskConsola`.

Es importante que, aunque no se vea en nuestro código, antes de bloquearse la tarea tiene que gastar un ciclo en el CPU, justamente para poder hacer la solicitud de *I/O*.

La figura 1 nos ayuda a entender mejor el funcionamiento de este tipo de tarea: los procesos 2 y 3 entran juntos en tiempo 0, el 0 y 1 en tiempo 5. Cada par corre en paralelo gracias a que hay dos procesadores, lo que facilita la comparación. El objetivo del primer par es ilustrar el caso en que la duración de las llamadas bloqueantes está prefijada: ambos procesos hacen dos llamadas de *I/O* cuya duración es exactamente 4 (para lograr esto hacemos que `bmin=bmax=4`). Por otra parte, los procesos 0 y 1, aunque reciben los mismos parámetros cada uno, muestran comportamientos distintos debido a que el rango de duración de cada bloqueo es de longitud mayor a 0.

Puede verse también que el tiempo que cada proceso pasa en el CPU antes de bloquearse es de exactamente un ciclo, que es lo que tarda en hacer la solicitud de *I/O*. En el último ciclo de cada proceso se hace el `exit`. Ambas cosas no dependen directamente de nuestro código, sino de las implementaciones de `uso_IO` y el simulador.

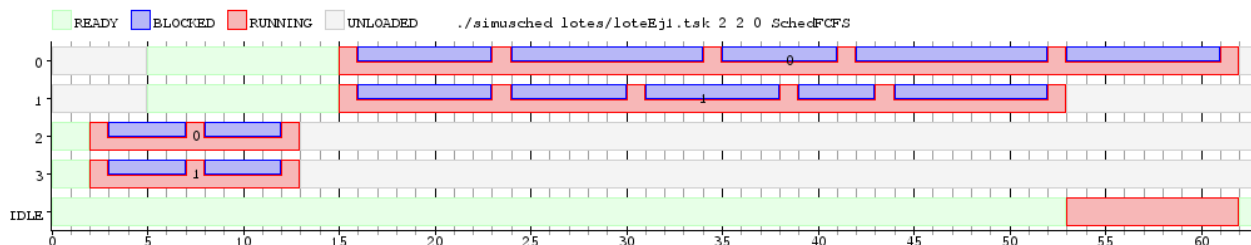
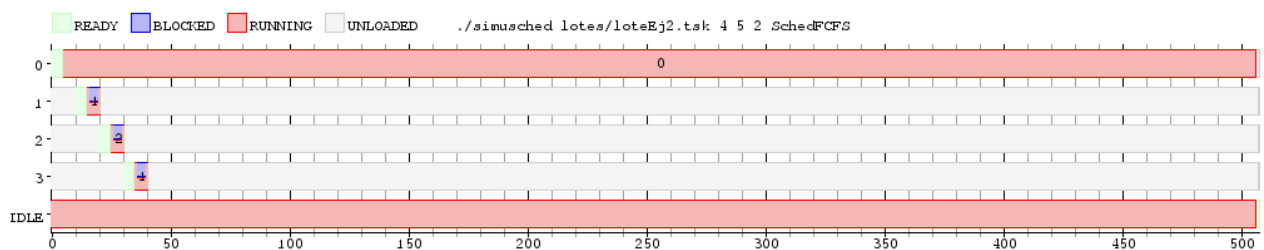
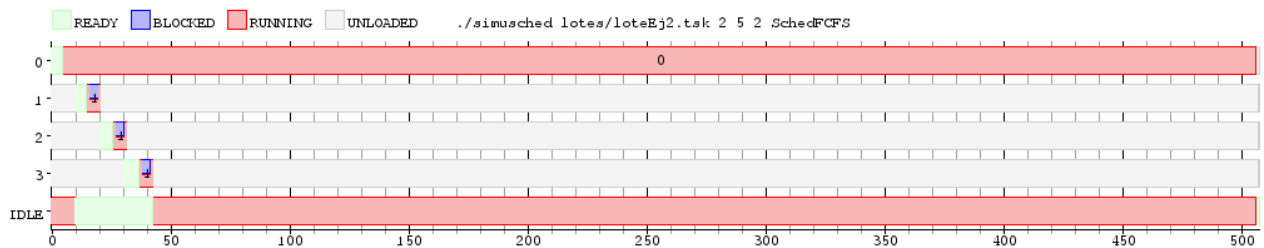
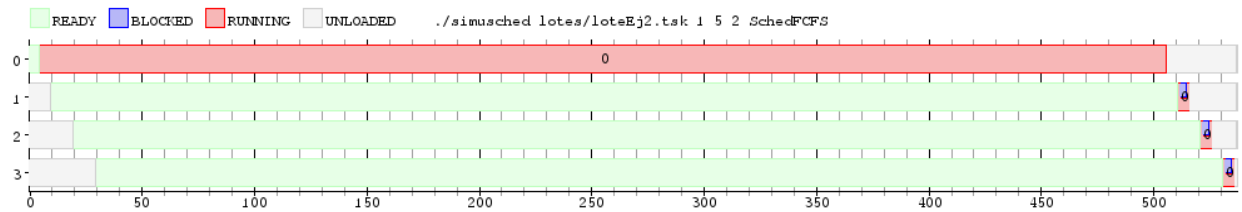


Figura 1



	Latencia		
#Procesadores	1	2	4
Tarea 0	5	5	5
Tarea 1	502	6	6
Tarea 2	503	7	6
Tarea 3	503	8	6
Promedio	378.25	6.5	5.75

y efectivamente responden (como puede apreciarse en la figura 2) genera para los usuarios la sensación de que “la máquina se colgó”. De hecho cabe la posibilidad de que si un usuario le pidió alguna información al sistema como parte de un protocolo, al momento de recibirlo esta deje de ser útil o válida.

Como contrapartida, vemos que el tener 4 procesadores no aporta grandes ventajas sobre tener 2, pues las diferencias de latencia son marginales como puede verse en la tabla, y de hecho uno de los procesadores se desperdicia completamente realizando la tarea *idle* como se aprecia en la figura 4.

3. Ejercicio 3

Creamos la función `TaskBatch` en el archivo `task.cpp`. Nuevamente la implementación es muy sencilla.

Tenemos una variable `tiempo_disponible`, que guarda el tiempo disponible de la tarea para ser usado en el CPU sin contar el tiempo que se necesita para lanzar las llamadas bloqueantes (el cual es de un ciclo por llamada).

Si los parámetros son coherentes, es decir que no se pide hacer más llamadas bloqueantes de las que es posible con el tiempo dado, entonces se itera sobre la cantidad de bloqueos, pasado por parámetro. En cada iteración se decide, de forma pseudo-aleatoria, cuánto del tiempo disponible del CPU se va a utilizar antes de hacer el respectivo bloqueo. Una vez usado el CPU por dicha cantidad de tiempo, se procede a actualizar `tiempo_disponible` y se lanza el pedido de *I/O*.

Una vez que se realizaron todos los bloqueos, se gasta en el CPU el tiempo disponible que pudiera haber sobrado (de no haberlo simplemente se prosigue con la finalización del proceso).

La figura 5 ilustra un lote de 4 tareas de este tipo.

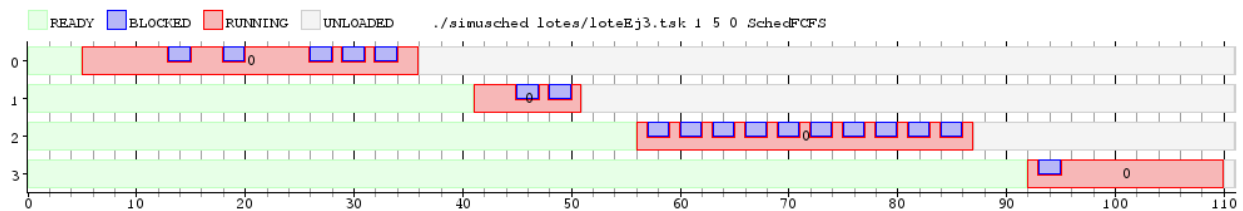


Figura 5

4. Ejercicio 4

A continuación detallamos los atributos privados de la clase **SchedRR**:

- **cant_cores**: almacena la cantidad de procesadores que tiene el sistema.
- **cola_procesos**: la cola *FIFO* en la cual se almacenan todos los procesos que están cargados, listos para ejecutar (o sea, en estado *ready*). Dicha cola es global para todos los procesadores.
- **quantum_original_cpu**: es un vector de longitud **cant_cores**, tal que **quantum_original_cpu[i]** indica la duración de un *quantum* del procesador *i*.
- **quantum_restante_cpu**: otro vector de longitud **cant_cores**, tal que **quantum_restante_cpu[i]** indica cuántos ciclos le quedan al proceso corriendo en el procesador *i* antes de agotar su *quantum*. En el caso en que se esté corriendo la tarea *idle* este valor no representa nada (pues tal tarea debe correr indefinidamente hasta que aparezca una nueva tarea para ser ejecutada).

Además, la clase cuenta con una función auxiliar, **int next(int cpu)**, que se encarga de devolver el *pid* del siguiente proceso a ejecutar, removiéndolo de la cola de procesos y reiniciando el *quantum* disponible para el proceso que llega. Notar que en caso de que no hayan procesos en *ready* los últimos dos pasos no se ejecutan y simplemente se devuelve el *pid* de la tarea *idle*.

La clase posee los siguientes métodos públicos:

- **void load(int pid)**: se encarga de cargar el proceso identificado por **pid**. Notar que esto simplemente consiste en agregarlo a la cola. Luego, eventualmente se ejecutará en un tick de reloj.
- **void unblock(int pid)**: vuelve a cargar una tarea que dejó de estar bloqueada, lo que consiste simplemente en llamar a la función **load**.
- **int tick(int cpu, const enum Motivo m)**: esta función se divide en tres casos según el motivo con el que se la haya llamado. Tanto en el caso en que la tarea que corría en **cpu** se haya bloqueado como en el que terminó hacemos lo mismo: sencillamente ponemos a correr a la siguiente tarea disponible (o a *idle* en caso de no haber ninguna), dejando a la tarea actual fuera del ciclo de ejecución (temporalmente en un caso, permanentemente en el otro). Si no sucedieron ninguna de las dos cosas entonces tenemos nuevamente tres posibles escenarios:

1. La tarea actual es *idle*, en cuyo caso solo queda llamar a **next** y devolver su resultado.
2. La tarea actual no es *idle* pero agotó su *quantum*, por lo que la volvemos a encolar (pues aún no ha terminado) y llamamos a **next**. Si no había otra tarea se seguirá ejecutando la misma durante otro *quantum*.
3. La tarea actual ni es *idle* ni terminó su *quantum*, así que debe seguir ejecutando pero reducimos en 1 la cantidad de ciclos restantes.

5. Ejercicio 5



Figura 6

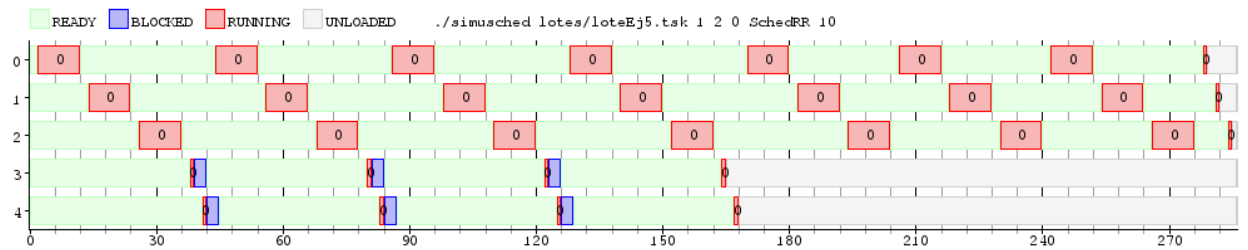


Figura 7

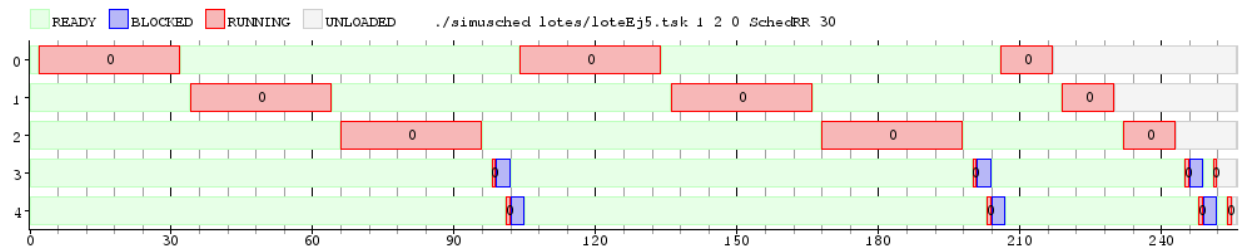


Figura 8

	Latencia		
Quantum	2	10	30
Tarea 0	2	2	2
Tarea 1	6	14	34
Tarea 2	10	26	66
Tarea 3	15	39	99
Tarea 4	18	42	101
Promedio	10.2	24.6	60.4

Tabla 2: Latencia de cada tarea y latencia promedio según la duración del *quantum* utilizado.

	<i>Waiting time</i>		
Quantum	2	10	30
Tarea 0	?	?	?
Tarea 1	?	?	?
Tarea 2	?	?	?
Tarea 3	?	?	?
Tarea 4	?	?	?
Promedio	?	?	?

Tabla 3: Tiempo total de ejecución de cada tarea (*turn-around*) y promedio según la duración del *quantum* utilizado.

	<i>Turn-around</i>		
Quantum	2	10	30
Tarea 0	447	279	217
Tarea 1	450	282	230
Tarea 2	453	285	243
Tarea 3	69	165	252
Tarea 4	72	168	255
Promedio	298.2	235.8	239.4

Tabla 4: Tiempo total de ejecución de cada tarea (*turn-around*) y promedio según la duración del *quantum* utilizado.

6. Ejercicio 6

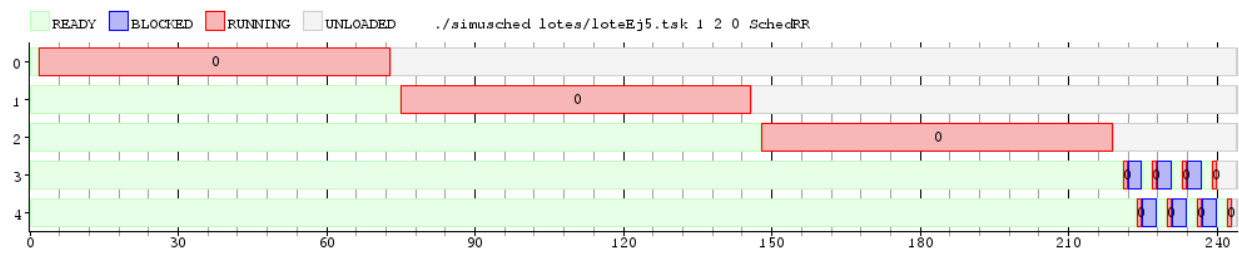


Figura 9

7. Ejercicio 7

8. Ejercicio 8