DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Sistemas operativos
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Federico De Rocco	403/13	fede.183@hotmail.com
Fernando Otero	424/11	fergabot@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 **2**
1.1. Desarrollo 2
1.2. Experimentación 2

2. Ejercicio 2 **3**
2.1. Desarrollo 3
2.2. Experimentación 3

3. Ejercicio 3 **4**
3.1. Desarrollo 4
3.2. Experimentación 4

4. Ejercicio 4 **5**
4.1. Desarrollo 5

5. Ejercicio 5 **6**
5.1. Desarrollo 6
5.2. Experimentación 6

6. Ejercicio 6 **7**
6.1. Desarrollo 7
6.2. Experimentación 7

7. Ejercicio 7 **8**
7.1. Desarrollo 8
7.2. Experimentación 8

8. Ejercicio 8 **9**
8.1. Desarrollo 9
8.2. Experimentación 9

9. Referencias **11**

1. Ejercicio 1

1.1. Desarrollo

Sabiendo que la tarea solamente se dedica a bloquearse una cantidad n de veces y con una duración al azar entre bmin y bmax, lo que hacemos para resolver este ejercicio es simplemente conseguirnos un número aleatorio n veces. Para esto usamos la función de c++ rand(), aclarando que queremos que los valores se encuentren entre los pedidos.

1.2. Experimentación

Para cumplir con lo pedido en el ejercicio utilizaremos el lote de tareas loteEj1.tsk, la representación del mismo usando la política FCFS es la siguiente:

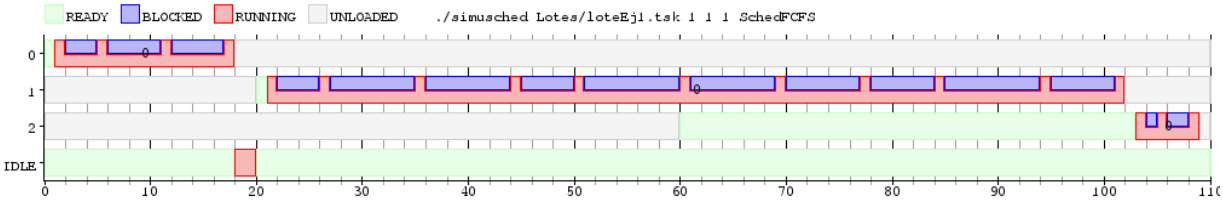

Figura 1: loteEj1.tsk con FCFS

Con este gráfico podemos ver como, para cada una de las tres tareas, van apareciendo varias llamadas bloqueantes de una duración diferente las unas de las otras. Considerando que la cantidad para cada una (especificada por el primer parámetro, n) es correcta y que el tiempo de las llamadas está entre los elegidos, podemos decir que el algoritmo es correcto.

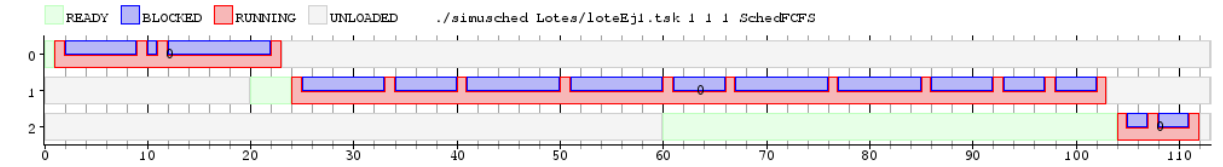

Figura 2: loteEj1.tsk con FCFS

Ejecutamos, por segunda vez, en las mismas condiciones para mostrar como varían los tiempos de los bloqueos.

2. Ejercicio 2

2.1. Desarrollo

2.2. Experimentación

3. Ejercicio 3

3.1. Desarrollo

Para poder resolver el problema de TaskBatch lo que hacemos es generar cant_bloqueos valores aleatorios no repetidos que no estén ubicados fuera de total_cpu. Lo generado por esto podría no estar ordenado, así que le hacemos un sort al resultado. Por último ciclaremos total_cpu veces y en las posiciones donde se tendría que bloquear hacemos la llamada usa_IO con duración de 1.

3.2. Experimentación

Siguiendo lo pedido por enunciado, realizaremos el gráfico del loteEj3.tsk usando el FCFS y nos queda lo siguiente:

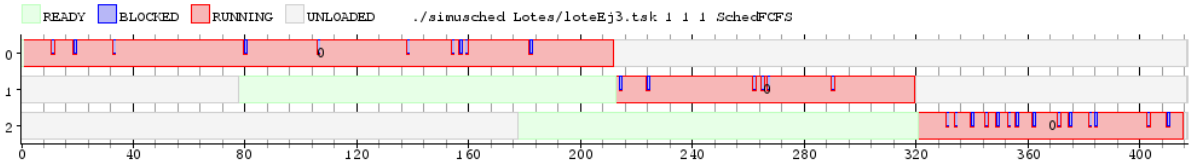

Figura 3: loteEj3.tsk con FCFS

4. Ejercicio 4

4.1. Desarrollo

El ejercicio consiste en programar un scheduler de Round-Robin, para esto utilizaremos las siguientes estructuras: vector de enteros pid_cores, vector de booleanos cores_bloqueados, vector de enteros quantum_restantes, entero cant_cores, entero cpu_quantum, Una cola de enteros enEspera.

Cuando creamos el scheduler se nos dan la cantidad de cores(que será cant_cores) y el quantum de los cpus(cpu_quantum). Segun el enunciado los procesos están agrupados en una única cola, de ahí viene enEspera. Como conocemos la cantidad de cores usamos tres estructuras de vectores para, solo teniendo el numero del cpu, poder acceder a la información de las mismas. cores_bloqueados nos indica si el proceso del núcleo en cuestión está o no bloqueado, pid_cores nos da el pid del proceso que esta corriendo en el cpu y quantum_restantes nos dice cuanto tiempo le queda por ejecutar hasta terminar. En el programa load en el cual tenemos que poner un proceso en el scheduler lo que haremos será buscar si hay algún core en pid_cores que tenga cargada la tarea IDLE.

Si lo encontramos, cargamos el proceso en ese core poniéndolo en pid_cores y dándole al quantum_restantes de ese proceso el valor de cpu_quantum ya que es una tarea que está por comenzar a ejecutar. Caso contrario, simplemente lo encolamos en enEspera para ser ejecutada cuando sea su turno. En el caso de unblock lo que hacemos es buscar entre los cores cual es el que tiene cargado el proceso que se desea bloquear(en pid_cores), una vez encontrado acceder a su posición en cores_bloqueados y asignarle true. Por último tenemos el programa tick que se encarga de realizar los procedimientos pertinentes en cada tick de reloj. Tenemos tres casos de motivos:

TICK: en el mismo sabemos que ha pasado un tick de reloj y debemos disminuir los cuantos que le quedan al procedimiento del cpu. En el caso de que estos terminen en 0 debemos plantearnos si debemos desalojar la tarea para poner otra. Si se da el caso de que el proceso está bloqueado entonces no realizamos ninguno de estos procesos ni disminuimos el quantum.

BLOCK: Disminuimos el quantum restante y bloqueamos el core(poniendo true en cores_bloqueados).

EXIT: Colocamos en el core la tarea IDLE y si la cola no está vacía, le asignamos la tarea que se encuentre próxima en ella. Después reseteamos los quantum del proceso.

5. Ejercicio 5

5.1. Desarrollo

5.2. Experimentación

6. Ejercicio 6

6.1. Desarrollo

6.2. Experimentación

7. Ejercicio 7

7.1. Desarrollo

7.2. Experimentación

8. Ejercicio 8

8.1. Desarrollo

Similar a lo visto en el ejercicio 8, tendremos que programar la estrutura de un scheduler de Round-Robin. La diferencia es que en este caso en vez de tener una cola única que vaya guardando todos los procesos, cada core tendrá su propia cola de procesos. Para resolver esto utilizaremos una estructura para representar el core que nos dará la siguiente información: un entero pidActual, un entero quantum_restante_actual, su cola de pids enEspera, un booleano que indica cuando está bloqueado.

Después tenemos la cantidad de cores y el quantum de los mismos. Por último necesitamos agrupar las estructuras core que tenemos. Para esto usamos simplemente un vector (nucleos).

La función load lo que hace es buscar cual de los cores tiene menor cantidad de procesos. Una vez encontrado, se pregunta si se está ejecutando la tarea IDLE entonces pone el pid en pidActual. Si no, lo encola en el enEspera del core.

unblock lo que hace es buscar cual es el core donde se encuentra el pid que se desea bloquear, una vez encontrado se asigna true a la variable bloqueado.

Por último tenemos la función tick. El código es equivalente al del ejercicio 4.

8.2. Experimentación

Según lo pedido debemos mencionar un caso donde la migración de nucleos sea beneficiosa y otro donde no. Para lo segundo, usaremos dos core y cuatro tareas que iran apereciendo en este orden: la más costosa, la menos costosa, la segunda más costosa y la segunda menos costosa. Usaremos, en los dos cores, un quantum de 20 y nos queda lo siguiente:

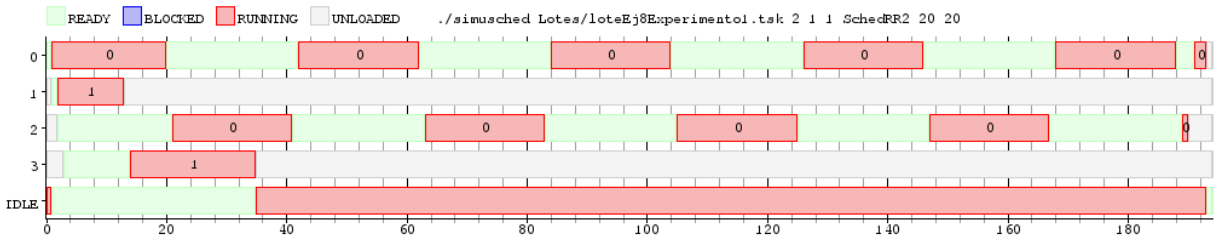

Figura 4: loteEj8Experimento1.tsk con RR2

Para el caso en el que sea beneficiosa vamos a usar el mismo lote de tareas, pero las cambiaremos de orden para que sean así: la más costosa, la segunda más costosa, la menos costosa y la segunda menos costosa. Nos queda lo siguiente:

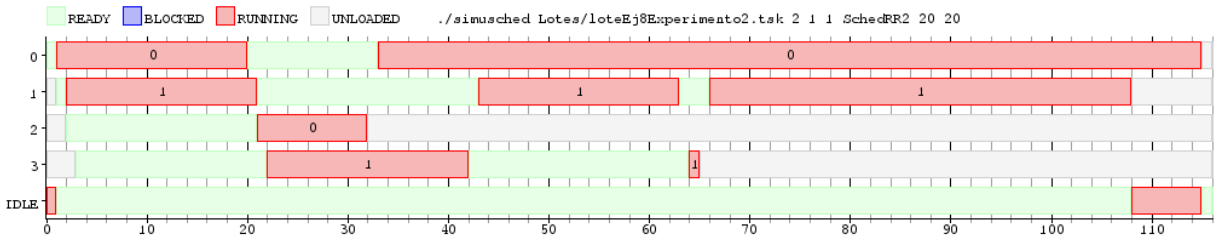

Figura 5: loteEj8Experimento2.tsk con RR2

Podemos ver que efectivamente el primer gráfico muestra que se tarda más en completar los procesos. Esto se debe a que el orden que le dimos a las tareas hace que el primer procesador se le asigne las tareas más largas y costosas, mientras que el segundo tiene las más cortas. En ese sentido se muestra como la mayor parte del trabajo recae en uno de ellos mientras que el otro termina y queda en ready mucho más rapido. Para el segundo experimento, al cambiar el orden podemos apreciar que los dos procesadores tienen una tarea muy costosa y otra no tanto. Efectivamente termina más rápido ya que el trabajo se distribuye de forma más justa.

9. Referencias

Referencias