DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Scheduling

Sistemas Operativos
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Confalonieri, Gisela Belén	511/11	gise_5291@yahoo.com.ar
Mignanelli, Alejandro Rubén	609/11	minga_titere@hotmail.com
Sabarros, Ian	661/11	iansden@live.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 - TaskConsola

Se programó un tipo de tarea llamado `TaskConsola`, que se ocupa de realizar *n* llamadas bloqueantes, cada una con una duración al azar comprendida entre los valores *bmin* y *bmax*.

1.1. Algoritmo

La Figura 1 muestra el pseudocódigo de esta tarea.

TASKCONSOLA$(n, bmin, bmax)$
1 **for** $i \leftarrow 0..n$
2 $ciclos\_bloqueo \leftarrow$ valor "random" en $[bmin, bmax]$
3 bloquear durante $ciclos\_bloqueo$ ciclos

Figura 1: Pseudocódigo TaskConsola

Para el cálculo del valor "random" se utilizó la función `rand()` provista por la librería `stdlib` de C++.

2. Ejercicio 2 - El lote de Rolando

Se escribió un lote de tareas para simular la siguiente situación:

- Correr un algoritmo que hace un uso intensivo de la CPU por 100 ciclos sin realizar llamadas bloqueantes.
- Escuchar una canción, que realiza 20 llamadas bloqueantes con una duración variable entre 2 y 4 ciclos.
- Navegar por internet, que realiza 25 llamadas bloqueantes con una duración variable entre 2 y 4 ciclos.

Estas tareas se ponen a correr en el instante 0, 1 y 2 respectivamente, y Rolando espera que ejecute11n en simultáneo.

2.1. Pruebas

Se ejecutó el simulador utilizando el algoritmo FCFS para 1 y 2 núcleos, con un cambio de contexto de 4 ciclos. Las Figuras 2 y 3 muestran el resultado de ambas simulaciones.

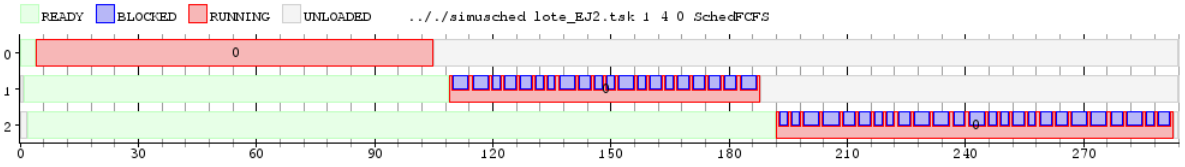

Figura 2: Simulación lote de Rolando con FCFS, 1 núcleo, 4 ciclos de cs

En el caso de la Figura 2, la latencia de cada proceso es:

- **Tarea 0:** latencia 4
- **Tarea 1:** latencia 108

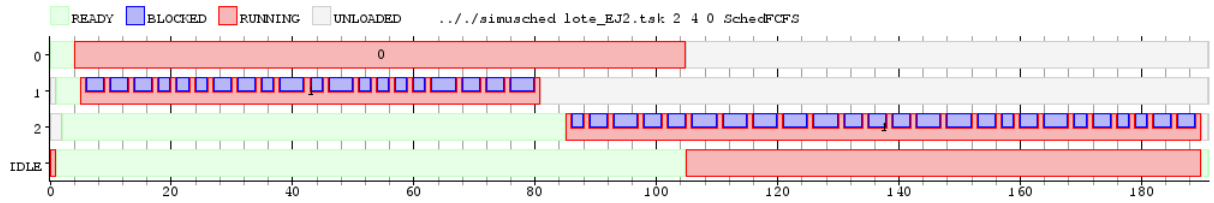

Figura 3: Simulación lote de Rolando con FCFS, 2 núcleos, 4 ciclos de cs

- **Tarea 2:** latencia 202

Estos valores nos dan un promedio aproximado de 30,3.

Por otro lado, en el caso de la Figura 3, la latencia de cada proceso es:

- **Tarea 0:** latencia 4
- **Tarea 1:** latencia 4
- **Tarea 2:** latencia 83

Estos valores nos dan un promedio aproximado de 100,6.

En el caso en el que Rolando se viera obligado a utilizar una computadora con un solo núcleo, como se utiliza un Sheduler del tipo FCFServe, no podría escuchar su canción preferida MIENTRAS corre el algoritmo, ya que las tres tareas se ejecutarán secuencialmente. En el caso de poder utilizar una computadora con dos núcleos, podría correrse el algoritmo en uno de ellos y la canción en el otro, y Rolando podría trabajar a gusto.

3. Ejercicio 3 - TaskBatch

Se programó un tipo de tarea llamado `TaskBatch`. Este tipo de tarea realiza *cant_bloqueos* llamadas bloqueantes en momentos elegidos pseudoaleatoriamente, y cada bloqueo dura 1 ciclo. Además, utiliza el el CPU durante *total_cpu* ciclos, incluyendo el tiempo necesario para lanzar las llamadas bloqueantes, pero no el tiempo en el que el proceso permanece bloqueado.

3.1. Algoritmo

La idea de nuestro algoritmo se basa en decidir, a cada ciclo y de manera pseudoaleatoria, si se realiza un bloqueo o no. Para tomar esta decisión vamos a tomar un valor entero “random” entre 0 y 1 (1 para bloquear y 0 para no hacerlo), nuevamente utilizando la función `rand()` de C++.

Como cada llamada bloqueante consume 1 ciclo de utilización de CPU, podemos decir que *total_cpu* incluye *cant_bloqueos* ciclos destinados a las llamadas bloqueantes, y el resto son usos “puros” de CPU. Por este motivo, la decisión pseudoaleatoria de bloquear la tarea se realizará $total\_cpu - cant\_bloqueos$ veces. Una vez realizados todos los usos “puros” de CPU, sólo resta realizar las llamadas bloqueantes necesarias para completar *cant_bloqueos*.

La Figura 4 muestra el pseudo-códgo de este algoritmo.

3.2. Pruebas

El lote que utilizamos consta de tres tareas, las mismas tratan de mostrar el comportamiento de nuestra `TaskBatch`. Las características de estas tareas son las siguientes:

TASKBATCH($total\_cpu, cant\_bloqueos$)

```
1  for i ← 0..(total_cpu − cant_bloqueos − 1)
2      bloquear ← valor “random” en [0,1]
3      if bloquear == 1∧ aún hay bloqueos por hacer
4          bloquear durante 1 ciclo
5          decrementar cant_bloqueos
6      else usar CPU durante 1 ciclo
7  while aún hay bloqueos por hacer
8      bloquear durante 1 ciclo
9      decrementar cant_bloqueos
```

Figura 4: Pseudocódigo TaskBatch

1. Posee sólo una llamada bloqueante, para mostrar que no siempre realizará la llamada bloqueante al final, sino que la ubicará en el medio con mayor probabilidad.
2. Posee muchas llamadas bloqueantes, de manera de que las mismas se realicen de forma consecutivas al final con mayor probabilidad.
3. Posee un número de llamadas bloqueantes cerca de un tercio de la cantidad de tiempo de cpu, para mostrar que las llamadas bloqueantes no necesariamente siguen un patrón.

La Figura 5 grafica el lote descrito utilizando un scheduler de tipo FCFS.

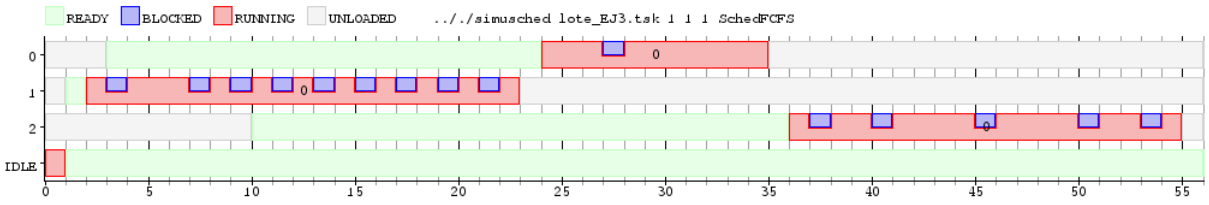

Figura 5: Simulación para TaskBatch, 1 núcleo, 1 ciclo de cs

4. Ejercicio 4 - Round Robin

4.1. Representación

Hemos representado las tareas con una estructura que contiene:

- un entero para el pid de la tarea
- un entero para el quantum restante de la tarea, en caso de que esté en estado *running*
- un booleano que indica si la tarea está bloqueada o no

Para implementar el scheduler, hemos utilizado los siguientes atributos:

- un arreglo de enteros de tamaño cantidad de cores utilizados, donde cada entero representa el quantum correspondiente a dicho core
- un arreglo de tareas del mismo tamaño, donde cada tarea representa la tarea que actualmente está corriendo en dicho core
- una lista de tareas, correspondiente a las tareas en estado *ready* y *waiting*

4.2. Funciones

Load Se crea una nueva tarea con el pid indicado y se agrega como último elemento de la lista de tareas *ready* y *waiting*.

Unblock Se recorre la lista de tareas *ready* y *waiting* hasta encontrar a la tarea con el pid indicado, y colocarla como "no bloqueada".

Tick Se cuenta con tres casos:

TICK Primeramente se decrementa el quantum restante de la tarea actual en el cpu indicado. Si aún tiene quantum para correr, se devuelve su pid. En caso de que se haya terminado su quantum, se evaluará si existe otra tarea en estado *ready*. De ser así, se devolverá el pid de la primer tarea que se encuentre en dicho estado en la lista de tareas. Si no hay otra tarea para correr, sea porque todas las demás se encuentran bloqueadas o porque no existen más tareas, se devuelve el pid de la tarea actual del cpu indicado.

BLOCK Se coloca la tarea actual del cpu indicado como "bloqueada", se la coloca al final de la lista de tareas y se procede a buscar la siguiente tarea a ejecutar. Si no hay más tareas o todas están bloqueadas, se devuelve la constante IDLE_TASK. En caso contrario, se devuelve el pid de la primer tarea en estado *ready* que se encuentre al recorrer la lista.

EXIT Si no hay más tareas o todas están bloqueadas, se devuelve la constante IDLE_TASK. En caso contrario, se devuelve el pid de la primer tarea en estado *ready* que se encuentre al recorrer la lista.

5. Ejercicio 5

6. Ejercicio 6

7. Ejercicio 7

8. Ejercicio 8