DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Scheduling

Sistemas Operativos

Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Arístides Catalano	279/10	dilbert_cata@hotmail.com
Laura Muiño	399/11	mmuino@dc.uba.ar
Jorge Quintana	344/11	jorge.quintana.81@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

0.1. Ejercicio 1 3
0.2. Ejercicio 2 4
0.3. Ejercicio 3 5
0.4. Ejercicio 4 6

0.1. Ejercicio 1

El ejercicio consiste en programar una tarea que simule n llamadas bloqueantes (con n pasado por parámetro) donde la duración de cada llamada bloqueante está determinada por los parámetros bmin y bmax.

Para generar las duraciones al azar dentro del rango [bmin, bmax], se utilizó la funcion rand de la libreria stdlib y se realizó lo siguiente:

int cant = bmax - bmin + 1;
int cant_ciclos_bloqueada;
cant_ciclos_bloqueada = bmin + (rand() % cant);

De esta forma se garantiza que *cant_ciclos_bloqueada* esté entre el rango pedido. Para realizar la simulación de las llamadas bloqueantes se ultilizó la función uso_IO provista por la cátedra.

Para probar la tarea se crearon los siguientes lotes:

- TaskConsola 5 10 20
- TaskConsola 15 2 20

En la primera tarea se eligieron esos parámetros para comprobar que la cantidad de tiempo bloqueado respetara los límites requeridos, dado que la primera simulación no permitía apreciar gráficamente a simple vista la variabilidad dentro del rango, se creo la segunda tarea, con un rango más amplio para que el diagrama de Gantt manifieste dicha variabilidad en los tiempos de bloqueo random de la tarea.

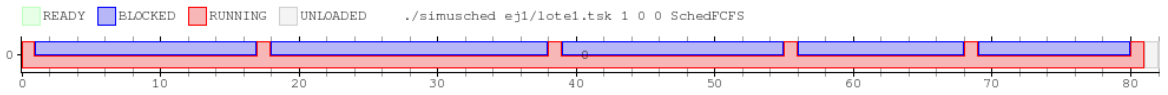

Figura 1:

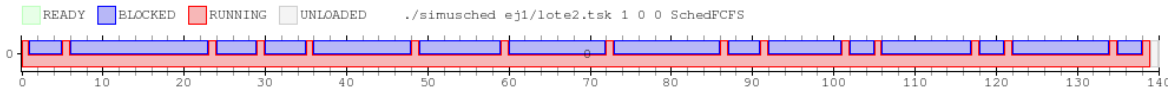

Figura 2: Figura 2.

0.2. Ejercicio 2

Para representar el uso de 100 ciclos de CPU y dos tareas bloqueantes definimos el siguiente lote de tareas:

TaskCPU
TaskConsola 20 2 4
TaskConsola 25 2 4

TaskConsola realizará 20 y 25 llamadas bloqueantes respectivamente, con cada bloqueo de duración variable entre 2 y 4 ciclos. El cambio de contexto se fijo a 4 ciclos. En este ejercicio, el enunciado no aclaraba nada de costos de migración, con lo cual lo mantuvimos en cero. Estos fueron los gráficos obtenidos :

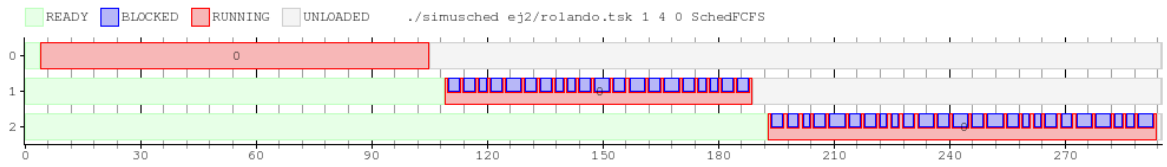

Figura 3:

La latencia es el tiempo que un proceso tarda en empezar a ejecutarse. Con un núcleo, la latencia para las tareas será la suma entre el costo de cambio de contexto y la duración de la tarea anterior. En el caso de la primer tarea (la cero) su latencia es 4. La segunda tarea (la uno) tendrá latencia $4 + 100 + 4 = 108$. Por último, la tarea 2 tendra latencia $108 + duracion\_tarea1 + 4$.

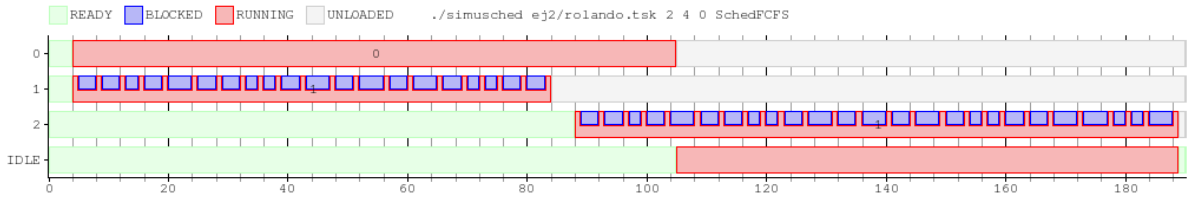

Figura 4:

En cambio, con dos núcleos, la tarea cero tiene latencia 4, y las tareas uno y dos tienen 4 y duracion_tarea1 + 4 respectivamente.

0.3. Ejercicio 3

Para realizar la tarea TaskBatch se generaron cant_bloqueos números distintos aleatoriamente dentro del rango [0, total_cpu -1]. Al hacerlo se tuvo en cuenta el hecho de que se pueden obtener números repetidos. Luego, si el momento coincidia con el número generado al azar, se llamaba a la funcion que simula los bloqueos (uso_IO).

La tarea generada es la siguiente

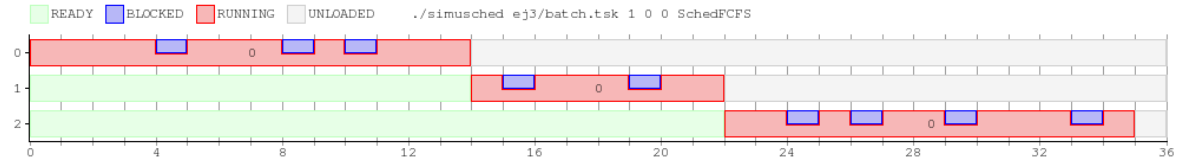

Figura 5:

0.4. Ejercicio 4

El ejercicio consiste en implementar el scheduler $Round-Robin$, que soporte migraciones entre núcleos y tenga una única cola global para los procesos.

Para esto se utiliza la estructura $queue < int >$, donde cada elemento de la misma representa un pid de algún proceso listo para ser ejecutado. También se debe guardar la cantidad de cores y sus respectivos quantums, que son pasados como parámetro cuando se llama al constructor de la clase. Se almacenan entonces estos datos y el quantum que le queda a cada proceso en cada CPU, usando dos estructuras de tipo $vector < int >$.

Los métodos a implementar son:
$load(pid)$: encola pid en $colaReady$ que es la cola de procesos en estado ready. $unblock(pid)$: hace la misma acción, encolar nuevamente el pid de la tarea que se desbloqueó. En el método $tick(cpu, motivo)$ se toman 3 casos: el proceso consumió todo el ciclo usando el CPU, realizó una llamada bloqueante o terminó de ejecutarse.

- TICK: Si la tarea era IDLE_TASK y no hay procesos en $colaReady$ sigue ejecutando lo mismo. Si la tarea era IDLE_TASK pero hay tareas disponibles en $colaReady$ se toma el tope de la cola (se quita de la cola), se actualiza el quantum disponibles y se devuelve ese pid. Si era cualquier otra tarea se resta un tick al quantum actual. Se chequea si se agotó su quantum, sino sigue ejecutando. Si lo consumió todo, se encola el pid en colaReady, se actualiza el quantum diponible, y se toma el tope de la cola como pid siguiente.
- BLOCK: Si no hay mas tareas en colaReady, se devuelve IDLE_TASK. Si hay, se actualiza el quantum diponible se devuelve el tope de colaReady quitando el elemento de la misma.
- EXIT: Ídem BLOCK.

0.5. Ejercicio 8

En el ejercicio 8 se pide realizar una implementación del scheduler round robin, pero sin migración de procesos entre núcleos. Para esto se agregaron nuevas estructuras a la clase SchedRR2: colaReady: ahora es un vector de colas, las cuales van a almacenar los pid de las tareas que ultilicen el core correspondiente. cantTasks: indica la cantidad de tareas que tiene cada cpu (RUNNING + BLOCKED + READY). cpuTareasBloqueadas: es una cola de tuplas con un pid en la primera coordenada (proceso que hizo llamada bloqueante) y número de core en el cual estaba ejecutando.

Los métodos load, unblock y tick también cambian respecto del round robin con migración de procesos. load(pid): ahora se encarga primero de buscar cual es el cpu con menor cantidad de tareas y, una vez encontrado, sumar uno a la cantTasks de ese cpu. Luego encola la tarea en la cola correspondiente de tareas ready. unblock(pid): como se pide que no haya migración de procesos entre núcleos, se guarda en la estructura *cpuTareasBloqueadas* todos las tareas bloqueadas con su número de core. Esta función encuentra la tupla en la cola, la quita de la cola, y luego encola la tarea en la colaReady del cpu obtenido. tick(cpu, m): Motivo TICK sigue siendo similar a la implementación anterior. Motivo BLOCK ademas de quitar al pid de la *colaReady* del cpu, arma una tupla de la forma (pid, cpu), siendo pid = current_pid(cpu). Luego encola esa tupla en *cpuTareasBloqueadas*, si quedan tareas en la *colaReady* toma la primera y devuelve su pid, y sino hay mas ready, devuelve IDLE_TASK. Motivo EXIT acá se resta uno a la cantTasks del cpu actual, y se devuelve la tarea que sigue en la *colaReady* y si no hay mas, IDLE_TASK.

Para corroborar su correcto funcionamiento se ideó un lote de tareas que permita chequear que, efectivamente no haya migración de núcleos y que se asignen correctamente los cores que cada tarea va a utilizar.

Lote utilizado:

TaskCPU 20 TaskConsola 20 1 5 TaskCPU 5 @2: *2 TaskCPU 20 TaskCPU 5 @4: TaskCPU 20 TaskConsola 10 1 5 TaskConsola 2 1 2 @18: TaskCPU 5 TaskConsola 3 2 4

La simulación se hizo con 3 cores con cambio de contexto 0, ya que no modifica el comportamiento del scheduler y costo de migración de núcleo 0. La idea del lote, es cargar los primeros cores con tareas que tarden mas en terminar, y dejar en el ultimo tareas que vayan terminando y que liberen el cpu, para poder apreciar de esta forma, que efectivamente se asignará el cpu con menos tareas. Debido a la implementación cuando todos los cores tengan igual cantidad de procesos, se le asignará el cpu mas chico numéricamente.

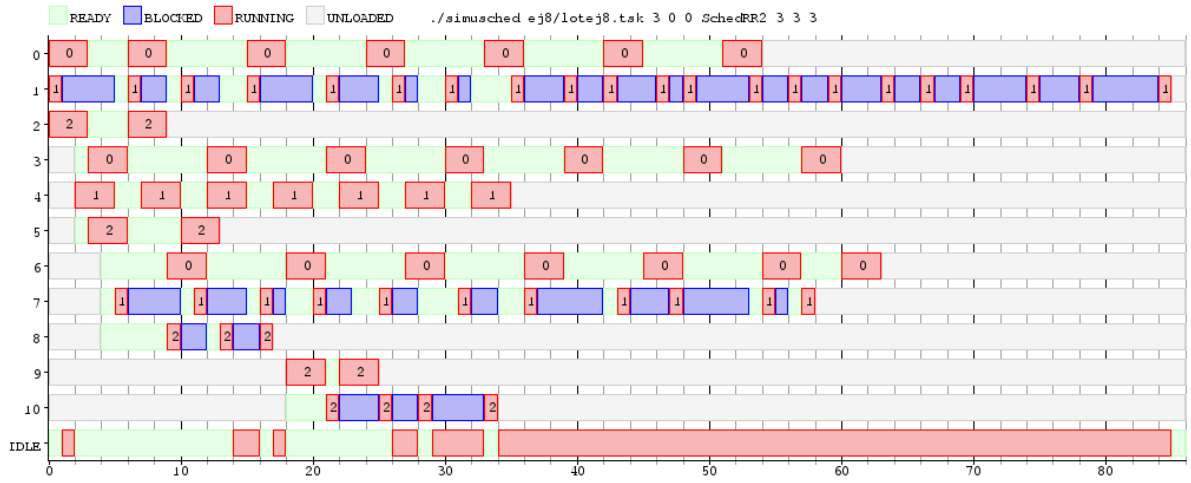

Figura 6:

Se puede observar que no existe migración de núcleos, ya que todas las tareas mantienen su número de cpu a lo largo de toda la simulación. En los instantes 0, 2 y 4 se ve que llegan las tareas y se les asignan los cores de la forma esperada. En el instante 4, los 3 cpu tienen 3 tareas cada uno. En el instante 15 el cpu 2 se queda sin tareas para ejecutar(la tarea 2 termina en el momento 2, la 5 en el 9 y la 13 en el 15).

por lo tanto se le asigna el IDLE_TASK por un ciclo de clock hasta que llegan dos nuevas tareas. Vemos que efectivamente se le asignan al core número 2, que es el que menos tareas tenia de los tres.