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

TP1 - Scheduling

Sistemas Operativos

Segundo Cuatrimestre de 2014

Apellido y Nombre	LU	E-mail
Cisneros Rodrigo	920/10	rodricis@hotmail.com
Rodríguez, Agustín	120/10	agustinrodriguez90@hotmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Contents

1 Introducción

En este Trabajo Práctico estudiaremos diversas implementaciones de algoritmos de scheduling. Haciendo uso de un simulador provisto por la cátedra podremos reprensentar el comportamiento de estos algoritmos. Implementaremos dos Round-Robin, uno que permite migración de tareas entre núcleos y otro que no y a través de experimentación intentaremos comparar ambos algoritmos. Asimismo, basándonos en un paper implementaremos una versión del algoritmo *Lotery* y mediante experimentos intentaremos comprobar ciertas propiedades que cumple el algoritmo.

2 Desarrollo y Resultados

2.1 Parte I – Entendiendo el simulador simusched

2.1.1 Introduccion

2.1.2 Ejercicios

- **Ejercicio 1** Programar un tipo de tarea TaskConsola, que simulara una tarea interactiva. La tarea debe realizar n llamadas bloqueantes, cada una de una duracion al azar 1 entre bmin y bmax (inclusive). La tarea debe recibir tres parametros: n, bmin y bmax (en ese orden) que seran interpretados como los tres elementos del vector de enteros que recibe la funcion.
- **Ejercicio 2** Escribir un lote de 3 tareas distintas: una intensiva en CPU y las otras dos de tipo interactivo (TaskConsola). Ejecutar y graficar la simulacion usando el algoritmo FCFS para 1, 2 y 3 nucleos.

Grafico algoritmo FCFS con 1 nucleo.

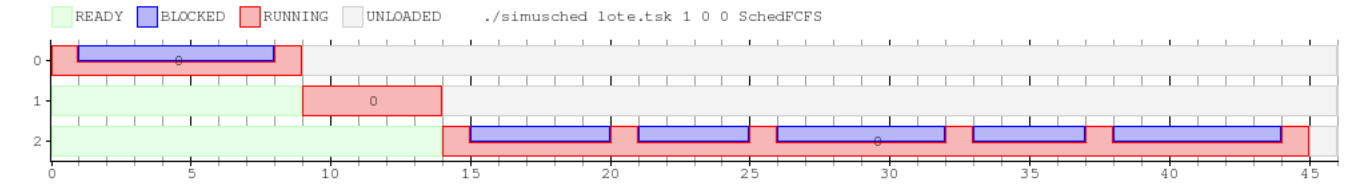


Grafico algoritmo FCFS con 2 nucleos.

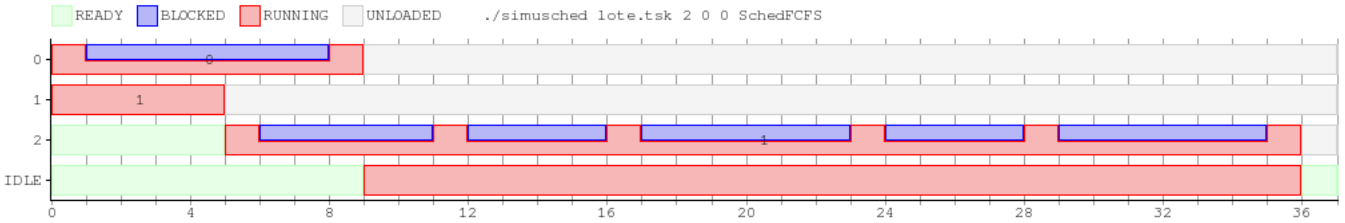
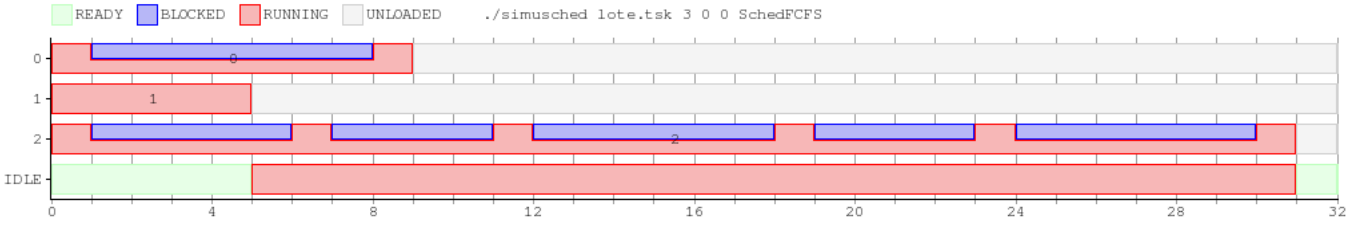


Grafico algoritmo FCFS con 3 nucleos.



Debido a las tareas de tipo TaskConsola, se observa que en los tres casos la duración de cada bloqueo es distinta.

A modo de análisis, se puede observar por medio de los gráficos como aumenta el paralelismo a mayor cantidad de núcleos. En este scheduler en particular esto ayuda de gran manera al rendimiento del sistema, puesto que un nucleo podrá ejecutar otra tarea recién cuando haya terminado la anterior. Las

consecuencias de este comportamiento son visibles en los 3 graficos. Agregando un core más, el tiempo que se tarda en ejecutar por completo todas las tareas de reduce casi a la mitad.

2.1.3 Resultados y Conclusiones

2.1.4 Ejercicio 1

```
 void TaskConsola(int pid, vector<int> params) {
int i, ciclos;

for (i = 0; i < params[0]; i++) {
ciclos = rand() % (params[2] - params[1] + 1) + params[1];
uso_IO(pid, ciclos);
}
}
```

QUEDA EXPLICAR EL CODIGO

2.2 Parte II: Extendiendo el simulador con nuevos schedulers

2.2.1 Introduccion

2.2.2 Ejercicios

- **Ejercicio 3** Completar la implementacion del scheduler Round-Robin implementando los metodos de la clase SchedRR en los archivos sched rr.cpp y sched rr.h. La implementacion recibe como primer parametro la cantidad de nucleos y a continuacion los valores de sus respectivos quantums. Debe utilizar una unica cola global, permitiendo ası la migracion de procesos entre nucleos.

- **Ejercicio 4** Diseñar uno o mas lotes de tareas para ejecutar con el algoritmo del ejercicio anterior. Graficar las simulaciones y comentarlas, justificando brevemente por que el comportamiento observado es efectivamente el esperable de un algoritmo Round-Robin.

- **Ejercicio 5** A partir del artıculo Waldspurger, C.A. and Weihl, W.E., Lottery scheduling: Flexible proportional-share re- source management. Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation – 1994. diseñar e implementar un scheduler basado en el esquema de loterıa. El constructor de la clase SchedLottery debe recibir dos parametros: el quantum y la semilla de la secuencia pseudoaleatoria (en ese orden). Interesa implementar al menos la idea basica del algoritmo y la optimizacion de tickets compensatorios (compensation tickets). Otras optimizaciones y refinamientos que propone el artıculo seran opcionales siempre que, en cada caso, se explique brevemente por que la optimizacion no se considero relevante a los efectos de este trabajo.

2.2.3 Resultados y Conclusiones

2.2.4 Ejercicio 3

Para completar la implementación del scheduler Round-Robin y que su comportamiento sea correcto, hicimos uso de diversas estructuras de datos cuya composición y uso describimos a continuación.

- Una cola global FIFO nombrada q, que contiene los pid de las tareas activas no bloqueadas y cuyo tope representa a la próxima tarea a correr. Utilizando una cola FIFO podemos obtener el comportamiento deseado, ya que al desalojarse una tarea por consumir su quantum esta misma será agregada nuevamente a la cola, quedando al final de ésta y generando el ciclo que buscamos. Al ser además la única cola para todos los cores, no se restringe a una tarea a ser ejecutada por un único núcleo, permitiendo así la migración entre nucleos.

- El vector *cores* contiene en su elemento i y el pid correspondiente a la tarea que está corriendo en el core $i + 1$. Inicializamos todos sus elementos en -1 (que se corresponde con la Idle Task) para reconocer que no se han cargado tareas en los núcleos de procesamiento.

- De la misma manera, el vector *quantum* contiene en la posición i el quantum definido para el núcleo y el vector *quantumActual*, contiene la cantidad de ticks que le quedan desde que se cargó la tarea en el core. En conjunto, ambas estructuras nos permiten determinar cuándo se consumió el quantum de una tarea, de manera tal que podamos desalojarla.

Ademas, tomamos ciertas decisiones en esta implementación las cuales detallamos a continuacion:

- Si una tarea se encuentra bloqueada cuando se produce el tick del reloj, esta misma es desalojada de la cola global, y agregada en un lista de *bloqueados*. A su vez, sera reseteado el quantum, se le dara inicio a la proxima tarea que se encuentre ready y cuando el sistema operativo, nos envie una señal de unblock, la tarea desalojada regresara al final de la cola global.

2.3 Parte 3: Evaluando los algoritmos de scheduling

2.3.1 Introduccion

2.3.2 Ejercicios

- **Ejercicio 6** Programar un tipo de tarea TaskBatch que reciba dos parametros: total cpu y cant bloqueos. Una tarea de este tipo debera realizar cant bloqueos llamadas bloqueantes, en momentos elegidos pseudoaleatoriamente. En cada tal ocasion, la tarea debera permanecer bloqueada durante exactamente un (1) ciclo de reloj. El tiempo de CPU total que utilice una tarea TaskBatch debera ser de total cpu ciclos de reloj (incluyendo el tiempo utilizado para lanzar las llamadas bloqueantes; no ası el tiempo en que la tarea permanezca bloqueada).

- **Ejercicio 7** Elegir al menos dos metricas diferentes, definirlas y explicar la semantica de su definicion. Diseñar un lote de tareas TaskBatch, todas ellas con igual uso de CPU, pero con diversas cantidades de bloqueos. Simular este lote utilizando el algoritmo SchedRR y una variedad apropiada de valores de quantum. Mantener fijo en un (1) ciclo de reloj el costo de cambio de contexto y dos (2) ciclos el de migracion. Deben variar la cantidad de nucleos de procesamiento. Para cada una de las metricas elegidas, concluir cual es el valor optimo de quantum a los efectos de dicha metrica.

- **Ejercicio 8** Implemente un scheduler Round-Robin que no permita la migracion de procesos entre nucleos (SchedRR2). La asignacion de CPU se debe realizar en el momento en que se produce la carga de un proceso (load). El nucleo correspondiente a un nuevo proceso sera aquel con menor cantidad de procesos activos totales (RUNNING + BLOCKED + READY). Diseñe y realice un conjunto de experimentos que permita evaluar comparativamente las dos implementaciones de Round-Robin.

- **Ejercicio 9** Diseñar y llevar a cabo un experimento que permita poner a prueba la ecuanimidad (fairness) del algoritmo SchedLottery implementado. Tener en cuenta que, debido al factor pseudoaleatorio involucrado, cualquier corrida puntual podrıa ser arbitrariamente injusta; sin embargo, si se repite un mismo experimento n veces y se observan los resultados acumulativos, tales anomalıas deberıan ir desapareciendo conforme n aumenta. En otras palabras, interesa mostrar en base a evidencia empırica que el algoritmo implementado efectivamente tiende a ser totalmente ecuanime a medida que n tiende a infinito.

- **Ejercicio 10** Los autores del artıculo sobre lottery scheduling alegan que la optimizacion de compensation tickets es necesaria para compensar una posible falencia del algoritmo inicial- mente propuesto en ciertos escenarios. Diseñar y llevar a cabo un experimento apropiado para comprobar esta afirmacion (provocar un escenario donde se manifieste el problema, comparar simulaciones ejecutadas con y sin compensation tickets y discutir los resultados obtenidos).

2.3.3 Resultados y Conclusiones

2.3.4 Ejercicio 6

```
  void TaskBatch(int pid, vector<int> params) {
int total_cpu = params[0];
int cant_bloqueos = params[1];
srand(time(NULL));

vector<bool> acciones = vector<bool>(total_cpu);

for(int i=0;i<(int)acciones.size();i++)
acciones[i] = false;

for(int i=0;i<cant_bloqueos;i++) {
int j = rand()%(acciones.size());
if(!acciones[j])
acciones[j] = true;
else
i--;
```

```
}

for(int i=0;i<(int)acciones.size();i++) {
if( acciones[i] )
uso_IO(pid,1);
else
uso_CPU(pid, 1);
}
}
```

QUEDA EXPLICAR EL CODIGO

2.3.5 Ejercicio 7

Las métricas elegidas fueron: NO ENCONTRE LA TEORICA POR AHORA CON ESTO -

2.3.6 Ejercicio 8

La idea central de esta version de Round-Robin es que no permita migración entre núcleos y esto se basa en utilizar una cola FIFO por cada núcleo, donde se encolarán aquellas tareas a las que se asignó el core correspondiente.

Para implementar este algoritmo, el Round-Robin 2, utilizamos varias estructuras. Estas son:

- Los vectores *quantum* y *quantumActual*, que cumplen la misma función que en nuestra implementación de Round-Robin.
- El vector de colas *colas*, donde en la posición i se encontrará la cola de tareas correspondiente a ese núcleo de procesamiento.
- El diccionario *bloqueados*, donde mantenemos aquellas tareas que se bloquearon con su número de core correspondiente y que nos permitirá, cuando la tarea se desbloquee, reubicarla en la cola del core que le corresponde.
- El vector de enteros *cantidad*, cuya única función será tener en la posición i la cantidad de tareas bloqueadas, activas o en estado ready que están asignadas al core i y que nos servirá para determinar a qué núcleo se asignará una tarea al momento de cargarla.

Cuando se carga una tarea, se chequea cuál es el core que menor cantidad de procesos activos totales tiene asignados(haciendo uso de la posición respectiva del vector *cantidad*). Una vez que se obtiene dicho núcleo, se agrega la tarea a la cola de tareas correspondiente al core y se actualiza la cantidad de tareas activas para ese núcleo sumándole uno.

Al bloquearse una tarea, se define una entrada en el diccionario *bloqueados* con el pid y el núcleo correspondiente. De esta manera, al desbloquearse, obtenemos el core en el que debe correr, eliminamos la entrada del diccionario y encolamos nuevamente el pid a la cola del núcleo en cuestión. De esta manera resolvemos parcialmente el problema de no permitir la migración entre cores.

Finalmente, cuando una tarea termina, se actualiza la cantidad correspondiente al núcleo restándole uno. Solamente a la hora de cargar la tarea y cuando una tarea termina se modifica dicha variable. De esta manera, aunque una tarea se bloquee seguirá reflejada en la cantidad del núcleo, lo que nos permitirá seguir el criterio que nos pidieron en la consigna a la hora de asignarle un core a la tarea.

3 Conclusión