Trabajo Práctico 1: Algoritmos de Scheduling
Grupo 10
Aronson, Alex
Ravasi, Nicolás

1. Introducción

En este trabajo tuvimos que evaluar el comportamiento de un Scheduler bajo distintas políticas de Scheduling. Para empezar se nos dio una implementación que soportaba múltiples Tasksets configurables. Todo el TP lo programamos usando el IDE Eclipse ya que consideramos que proporcionaba una manipulación más sencilla. En esta entrega adjuntamos solamente los sources *.java* contenidos en la carpeta */src/sisop*, más este PDF obviamente.

2. Políticas sin preemption

En primer lugar, debíamos, analizando dos diagramas de Gantt correspondientes a diferentes planificaciones para un mismo taskset, para esto analizamos los resultados producidos por cada tipo de scheduler y dedujimos cuánto debían durar las tareas y en qué orden debían llegar para que el output generado por el programa sea el pedido.

Luego se nos pedía modificar las clases FCFS y SJF dadas para que outputearan el waiting time al ser ejecutadas, esto lo logramos haciendo que a cada tarea en la cola de listas del scheduler (o sea, las que llegaron, no terminaron y tampoco se están ejecutando) se le sumara un tiempo en cada iteración del mismo; una vez terminado, simplemente pedíamos que se calculara el promedio entre la suma de todos los waiting times y la cantidad de tareas.

Con esto implementado, debemos analizar el comportamiento de estas dos políticas. A continuación presentamos los resultados obtenidos

TaskSet	Scheduler FCFS	Scheduler SJF
ts1	105	87
ts2	210	122
ts3	127	105
ts4	127	58
ts5	144	56
ts6	176	57

Tabla 1, comparación del waiting time promedio entre FCFS y SJF

Salta a la vista la mejor performance del algoritmo SJF por sobre el FCFS. Esto es algo evidente, porque el SJF produce waiting times muy bajos para los trabajos más cortos que llegan al principio, aún cuando otros más largos hubieran llegado al mismo tiempo pero antes en la lista. O sea, el SJF es un scheduler goloso, en cada momento elige la mejor opción, esto es, el proceso más corto que es el que le permite minimizar el waiting time general. La mayor desventaja del SJF es, se sabe, la inanición que produce en los procesos más largos, ya que estos pueden estar mucho tiempo inactivos si siguen llegando procesos más cortos, el FCFS no tiene esta desventaja ya que su asignación de procesador se basa en quién llegó antes, independientemente de su costo; esto es bueno para asegurar una equitatividad pero malo para el rendimiento global.

3. Políticas con preemption

Luego de esto, el trabajo se centraba en el estudio de políticas de scheduling con desalojo. Para esto, debíamos implementar los algoritmos para Round Robin y Multilevel Feedback Queue. Para hace esto, primero realizamos un análisis sobre qué era lo que debíamos tomar en cuenta. La implementación de ambas es bastante similar, ya que se basan en el mismo principio. Primero, debíamos analizar qué era lo que implicaba un cambio de contexto. Así es que decidimos que este cambio (con su consecuente retardo de dos unidades de tiempo) se produciría al entrar en el procesador una tarea que ya había sido ejecutada previamente, o sea, cuyo *ttime* fuera mayor a 0, o al salir una que no hubiera terminado (si estas dos condiciones se daban, sólo sería un cambio de contexto, no dos). No consideramos context switch si al terminar una tarea comenzaba una nueva que no hubiera sido ejecutada, por otra parte. Para hacer el context switch simplemente aumentábamos un contador y devolvíamos esa frase, al entrar, chequeando el contador el algoritmo mismo se encargaría primero de devolver un nuevo context switch como resultado para la segunda unidad de tiempo, y seguidamente escoger cuál es la tarea que le toca ejecutar a continuación.

Para programar el algoritmo Round Robin, utilizamos una cola a la que llamamos *robins*, el tope de la cola es la tarea que se está ejecutando, al acabarse el quantum se popea el tope y se lo agrega al final de la pila, en ese momento decidimos si llamamos a un context switch o no. Al final de la ejecución de ese instante de tiempo, se le agrega un tiempo de waiting time a todas las tareas no ejecutándose, esto lo podemos hacer ya que recorremos toda la lista de pendientes, si no corresponden con la ejecutándose o terminadas, están esperando.

Una vez terminado el algoritmo y funcionando, podíamos modificar el quantum del mismo para verificar el rendimiento hasta encontrar un óptimo, ponemos a continuación los resultados obtenidos

TaskSet	Quantum = 5	Quantum = 10	Quantum = 20	Quantum = 30	Quantum = 50
ts1	253	190	160	138	109
ts2	307	253	241	240	237
ts3	293	228	198	186	149
ts4	209	147	128	108	99
ts5	153	125	118	102	109
ts6	150	125	127	119	130

Tabla 2, comparación del waiting time para diferentes quanta del algoritmo RR

Puede observarse que el quantum default de 5 está lejos de ser el ideal, ya que es muy corto y por ende provoca demasiados cambios de contexto. El ideal (o sea, el que produce un tiempo de espera promedio menor, y por ende mejor turnaround) está entre 30 y 50, no demasiado corto para provocar más context switches de lo necesario, ni demasiado largo para provocar inanición para las demás tareas al mantener a una ejecutando por mucho tiempo y por ende tener demasiadas en ready que acumulen mucho *wtime*.

Para la Multilevel Feedback Queue, el mecanismo es similar, la diferencia radica en que en vez de haber una sola cola con todas las tareas listas hay tres, de prioridades alta, media y baja (*robinh*, *robinm* y *robinl*, respectivamente). Al ser desalojada una tarea de una cola, va al fondo de la inmediatamente inferior (si estaba en *robinl*, permanece en esa). Para decidir la nueva tarea a ejecutar, el algoritmo simplemente recorre las colas en orden, si no encuentra nada en la de alta prioridad se mueve hacia la media y luego hacia la baja. Si bien el funcionamiento es bastante simple, la cantidad de casos borde era grande, por lo que hubo que considerar todos esos casos, lo que explica la gran cantidad de ifs que tiene el algoritmo.

Luego se nos pidió que modificáramos el algoritmo de la MFQ para que soportara una pseudo-simulación de entrada/salida, esto es, procesos que no utilizan todo su quantum porque se bloquean esperando I/O. Para hacer esto, consideramos en primer término que si modificábamos la clase MFQ no se vería claramente lo implementado para el ejercicio 6, por lo tanto decidimos copiar el código a una nueva clase ($MFQIO.java$) y hacer la nueva implementación ahí, esto requirió que también cambiáramos la clase $Simulator.java$ para que soportara una política más de Scheduling. A la hora de la implementación recurrimos a los números aleatorios, al momento de sumar ttime al proceso, en vez de hacerlo sin más se llama a dos generadores random, uno da un resultado que es 1 ó 2, tomamos 1 como si fuera "*true*", lo que significa que puede hacer

I/O, en tal caso se observa al segundo número aleatorio, entre 1 y 100, éste nos fijamos si es mayor que el porcentaje de quantum ya consumido, en caso afirmativo, consideramos que se produce una llamada a I/O y por lo tanto desalojamos al proceso colocándolo en el fondo de la cola de alta prioridad. Consideramos que las dos unidades de tiempo causadas por el context switch son las necesarias para la entrada/salida, por lo tanto, aún si las colas están vacías y le tocaría inmediamente a este proceso nuevamente, tomamos esas 2 unidades como costo en el context switch.