Sistemas Operativos II

Práctica 3

Daniel Rivero
Dennis Quitaquís

Introducción

El objetivo de esta práctica es, basándonos en el código implementado en la práctica 2, conseguir que la función que se encarga de crear el árbol utilice más de un hilo de ejecución para reducir el tiempo que tarda en hacerlo.

Para ello utilizaremos funciones que se encargarán de crear y gestionar hilos de ejecución y, también, implementaremos monitores para sincronizar correctamente los hilos y conseguir exclusión mutua para evitar problemas.

Memoria de la práctica

La función que crear el árbol que hemos implementado funciona, a grandes rasgos, de la siguiente manera:

- El hilo principal del programa se encarga de pedir el archivo que contiene la lista de ficheros a analizar.

- Lee la primera línea para saber cuantos ficheros hay y crea los hilos indicados por la variable global MAXTHREADS con la función pthread_create, pasándole la función que ejecutarán los hilos (thr_fn) y los argumentos necesarios (void *)par); y los hilos se empezarán a ejecutar. El hilo principal esperará a que acaben todos los hilos creados antes de finalizar él (usando la función pthread_join).

- Cada hilo, mientras aún queden ficheros por analizar, cogerá el siguiente fichero de la lista (esta parte estará protegida con un pthread_mutex_lock(&mutex) para conseguir la exclusión mutua y que no haya conflictos con los ficheros); a continuación, el hilo creará una tabla hash con las palabras del fichero que analice. Por último, el hilo insertará todas las palabras de la tabla hash en el árbol (esto se protegerá con otro monitor usando pthread_mutex_lock(&mutex2) para evitar conflictos al modificar el árbol).

- El procedimiento anterior se repetirá hasta que no haya más ficheros por analizar, momento en que los hilos finalizarán su ejecución y el hilo principal del programa se despertará y volverá al menú principal de la aplicación.

Para comprobar el correcto funcionamiento de la creación del árbol usando múltiples hilos la hemos ejecutado con diversas cantidades de archivos y diferente número de hilos y ha funcionado correctamente, así como su ejecución con Valgrind que indica que no hay problemas de memoria sin liberar.

Como ejemplo, en la carpeta Proves hay un fichero log.txt con el log de la ejecución del programa (toda la impresión por pantalla que hace el programa según se ejecuta) y un fichero valgrind.txt con el análisis hecho por valgrind que demuestra que no hay memoria no liberada; estas pruebas se han hecho en un ordenador con 2 procesadores, utilizando 4 hilos y analizando 50 ficheros.

Además, hemos hecho una comparativa con los tiempos de ejecución (de CPU y cronológico) según la cantidad de hilos usados. Estas pruebas se han realizado en un ordenador con procesador i7 de 8 núcleos a 3.2GHz, analizando 100 ficheros.

Hilos	**1**	**2**	**4**	**8**	**16**	**32**
Tiempo CPU (s)	20,40	21,02	24,87	45,51	45,47	53,01
Tiempo Cronológico (s)	20,47	10,60	6,41	7,25	10,09	11,98

Con esta comparativa podemos ver que hasta cierto número de hilos se va creando el árbol más rápido, pero llega una cantidad de hilos a partir de la cual se tarda más. Esto es debido a que hay un momento en que, por muchos hilos que crees, los hilos perderán más tiempo esperando para coger la clave del monitor (para poder coger el archivo a leer o para pasar la tabla hash al árbol) que haciendo cosas, por lo que dejará de ser óptimo.

Ese problema también habrá que tenerlo en cuenta a la hora de elegir a qué nivel se debe implementar la exclusión mutua; en nuestro caso, si debemos proteger toda la función que copia los datos de la tabla hash al árbol o únicamente las funciones específicas que manipulan el árbol.

En nuestro caso, hemos optado por proteger toda la función ya que al intentar proteger únicamente las funciones específicas, al haber tanto lock y unlock en cada función por parte de los hilos, estaban más tiempo esperando a poder coger la clave que ejecutándose activamente por lo que dejaba de ser efectivo y tardaba más tiempo que protegiendo la función entera.