Programación Concurrente
Trabajo práctico final

Año: 2021

Cuatrimestre: 1º cuatrimestre

Profesores:

- **Daniel Ciolek**
- **Pablo Terlisky**

Integrantes

Nombre y Apellido:	Nahuel Gómez	Matías N. Fuentes
Mail:	nahuelggt@gmail.com	matiasnfuentes@gmail.com
Legajo Nº:	35.056	48.423

Introducción

Siguiendo las especificaciones técnicas y el modelo base dado, buscamos realizar una solución eficiente y funcional para poder cumplir con ellos.

Nos decidimos por una implementación en la que tomáramos como tarea cada uno de los cien hashes del archivo a decodificar, ya que nos parecía una unidad de trabajo que permitiría distribuir la carga del mismo entre los thread de forma bastante equitativa. Por otro lado, esto permite un flujo de trabajo continuo y minimiza la carga sobre el buffer.

Nuestro método main es el encargado de instanciar un buffer, para alocar las tareas, con un tamaño pasado por parámetro.

Implementamos el mismo de forma muy similar a lo que vimos en la teórica, utilizando un array como estructura de datos (en este caso de Runnable), y haciendo que la lectura y escritura sobre el mismo sean métodos synchronized para garantizar que cada thread podrá tomar una tarea de este en exclusión mutua.

Otra de las funciones de nuestro método main es instanciar un threadpool que tiene como parámetros el buffer, y la cantidad de threads con la cual se quiere trabajar.

Este threadpool será el encargado de instanciar los threads que necesitemos y también de ponerlos a correr. Además, cuenta con los métodos launch y stop:

- Launch: permite poner una tarea en el buffer para que sea ejecutada por los threads.
- Stop: lanza tantas tareas PoisonPill como threads instanciados haya. Cuando estas tareas son ejecutadas por los workers, se lanza una excepción y su ejecución se detiene.

Una vez que los thread están corriendo, es el momento de lanzar las tareas de decodificación. Para esto tenemos la clase “Reader” que se encarga de leer los hashes a decodificar de un archivo, y retornárnoslos uno a uno. Leído el hash, se crea y se escribe una “DecodeTask” en el buffer, que toma como parámetro el hash, el salt máximo (también pasado por parámetro al main) y el path al diccionario que queremos utilizar. Este proceso es repetido hasta agotar todos los hashes del archivo “passwd.txt”.

Terminada esta etapa se llama al método stop del threadpool, para que lance las PoisonPill que sean necesarias teniendo en cuenta que, una vez terminadas las tareas de decodificación, todos los threads deben tomar una de ellas y finalizar su ejecución.

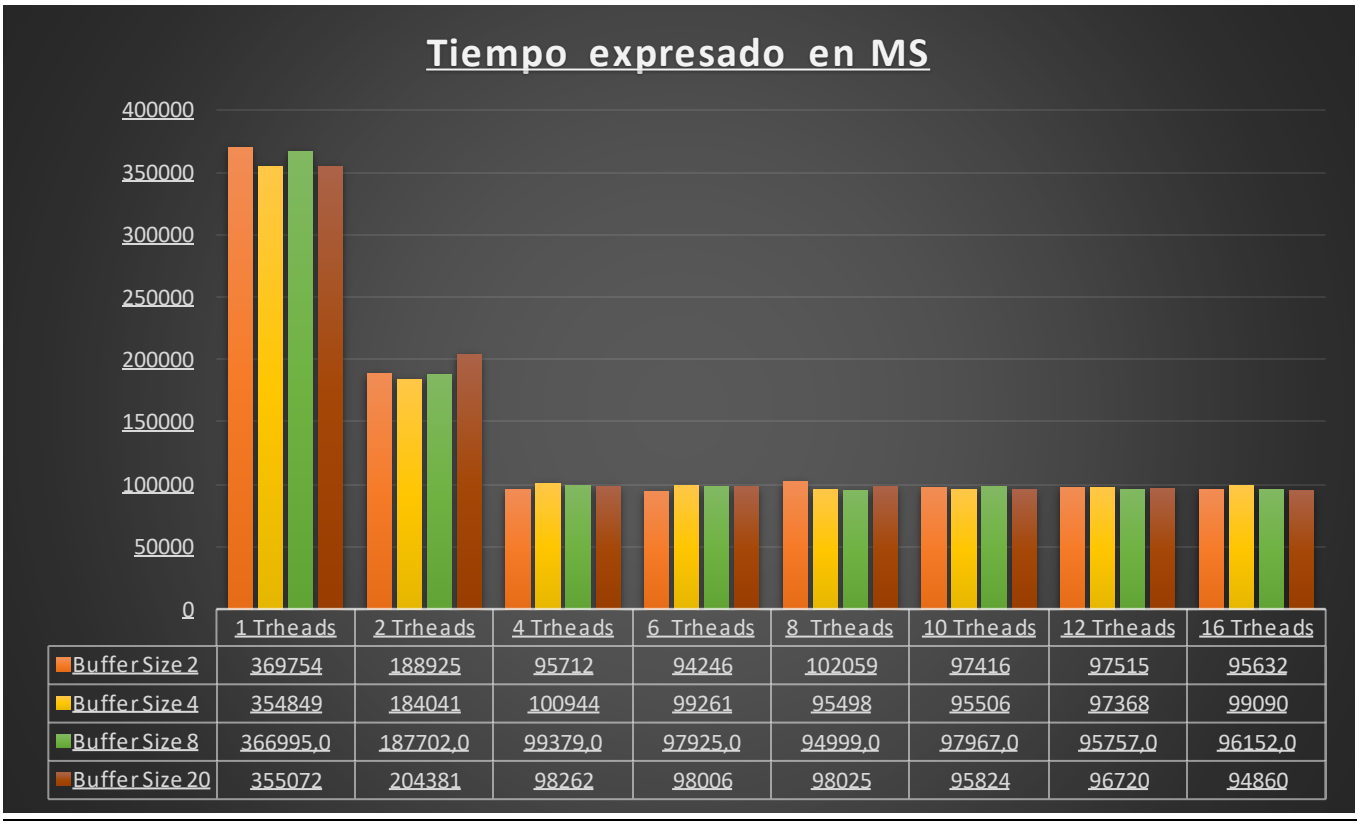
Nuestra DecodeTask consiste en formar con cada uno de los salt y los posibles passwords dados un hash, y verificar si este coincide con el que buscamos. A penas se obtiene el hash buscado, se sale del ciclo (no se prueban más combinaciones) y se escribe el password asociado a ese hash, en el archivo cracked.txt.

Para hacer la escritura de este, utilizamos la clase archivo de salida. La misma tiene el método sychronized write, que permite hacer la escritura de las contraseñas decodificadas en exclusión mutua, garantizando de esta forma que no vamos a perder ninguna de ellos cuando los threads llamen a la escritura.

Evaluación

Las pruebas fueron ejecutadas en un equipo que constaba de :

- Ryzen 3 3200g de 4 núcleos y 4 Threads
- 8 gb de ram 2666mhz
- SSD Kingston A400 SATA III (Hasta 545 MB/S lectura y escritura)



	1 Trheads	2 Trheads	4 Trheads	6 Trheads	8 Trheads	10 Trheads	12 Trheads	16 Trheads
Buffer Size 2	369754	188925	95712	94246	102059	97416	97515	95632
Buffer Size 4	354849	184041	100944	99261	95498	95506	97368	99090
Buffer Size 8	366995,0	187702,0	99379,0	97925,0	94999,0	97967,0	95757,0	96152,0
Buffer Size 20	355072	204381	98262	98006	98025	95824	96720	94860

Análisis

Luego de realizar la ejecución del programa repetidas veces y de analizar el grafico presentado llegamos a la conclusión de que, el tiempo de ejecución va a oscilar dependiendo del hardware y de la cantidad de threads reales que tenga el procesador. Cuando los threads que lanzamos en el programa son mayores a la cantidad de trheads reales del procesador los tiempos empiezan a ser similares, y el tamaño del buffer pasa a ser algo secundario, que no influye tanto en la ejecución.

En cambio, entre uno y cuatro threads con el mismo tamaño de buffer, los tiempos de ejecución cambian y mucho, dado que con uno o dos threads no se aprovecha el límite del procesador. Por otro lado, en nuestra implementación el tamaño del buffer no pareciera ser algo que influya drásticamente en las ejecuciones. Esto se debe a que nuestro main va a estar constantemente intentando poner una tarea en el buffer, para que uno de los threads la pueda ejecutar, y el tiempo que se pierde en sacar una tarea del buffer, y agregar una nueva al mismo para que un thread la consuma no es significativa. Es decir, a grandes rasgos, nuestros threads pasaran un tiempo ocioso (sin ejecutar) relativamente insignificante comparado con el tiempo total de ejecución, por mas que el tamaño del buffer sea el más pequeño.

Si tuviéramos una implementación del buffer distinta, con operaciones de escritura y lectura que sean mas costosas, probablemente este tuviera mas incidencia en el tiempo final de ejecución.

Otra cuestión a tener en cuenta es que, si hubiéramos tomado como unidad de tarea cada uno de los passwords del diccionario a probar, en vez de los hashes a decodificar, también el rol del buffer hubiese sido mucho mas importante en los tiempos finales.

Como cuestión final para agregar, y no menos importante, destacamos que CONCURRENCIA no es PARALELISMO. Como mencionamos en la primera clase de la materia, la concurrencia es un paralelismo potencial. Como los tiempos de procesamiento hoy en día son tan rápidos, tenemos la sensación de que los programas se están ejecutando todos al mismo tiempo. Pero la realidad es que , si tenemos un solo procesador real, solo un programa está ejecutando a la vez. El paralelismo en cambio, lo podemos ver cuando tenemos ejecutando cuatro programas en cuatro núcleos distintos. En ese caso, si vemos una notoria mejora en los tiempos de ejecución. Si nuestro programa tuviera muchas operaciones bloqueantes, la concurrencia sería un factor CLAVE para acelerar los tiempos de ejecución , y en ese

caso, tener lanzar más threads que se ocupen concurrentemente del programa , si sería una solución muy recomendable.

Esto nos lleva a aprender que lanzar threads por lanzarlos , no va a ser que nuestra solución sea mas eficiente o mejor. Como profesionales debemos tener el suficiente criterio para analizar cómo trabaja nuestro programa , y ver si ejecutarlo con distintos threads es necesario, ya sea porque mejora nuestra eficiencia, o porque en necesario utilizar dicho programa por varios usuarios concurrentemente (sistemas inherentemente concurrentes).