Programación Eficiente

Alumno: Jasin Anibal.

Tema: Profiling 2

Fecha entrega: 20/09/2018

Libera memoria correctamente

Función BorrarMatriz()

La función comienza liberando memoria desde la posición 1 en la matriz, pero debería hacerlo desde la posición 0, porque todo array comienza en 0.
Para corregirlo Inicio variable i=0. Al final, también hace falta liberar la variable que apunta a la matriz.

Función CrearMatriz()

En esta función, al crearse la matriz se hace un ciclo y en cada vuelta se va creando un array de punteros, pero se observa que para crear n filas el ciclo realiza n + 1 iteraciones, por lo tanto crea un fila demás que no es utilizada y sobra. Entonces corregimos ese ciclo, haciendo que cree solo las filas necesarias.

Para una matriz de 4000 x 4000, comparamos los resultados

En el caso sin corregir vemos que tenemos 2 bloques de 64,000 bytes que definitivamente se perdieron. A partir de estos indirectamente se perdieron 4000 bloques de 128.000.000 bytes.

```
HEAP SUMMARY:
in use at exit: 128,128,000 bytes in 4,004 blocks
total heap usage: 8,004 allocs, 4,000 frees,
256,128,000 bytes allocated
```

```
LEAK SUMMARY:
definitely lost: 64,000 bytes in 2 blocks
indirectly lost: 128,000,000 bytes in 4,000 blocks
possibly lost: 64,000 bytes in 2 blocks
still reachable: 0 bytes in 0 blocks
suppressed: 0 bytes in 0 blocks
Rerun with --leak-check=full to see details of leaked memory
```

El caso corregido, se ve como se pudo liberar correctamente la matriz de la memoria:

```
HEAP SUMMARY:
in use at exit: 0 bytes in 0 blocks
total heap usage: 8,002 allocs, 8,002 frees, 256,064,000 bytes allocated

All heap blocks were freed -- no leaks are possible
```

Es eficiente en cuanto a tiempo de ejecución y acceso a memoria.

En la función CargarMatriz() y MostrarMatriz(), se observa que se recorre la matriz analizando un elemento de cada columna por vuelta. Esto genera ineficiencias en tiempo de ejecución y acceso a la memoria, porque el lenguaje C al ser row-major, los elementos de la misma fila se almacenan consecutivos en la memoria RAM unos a

otros. Entonces al recorrer la matriz de manera que en cada ciclo se accede a elementos no contiguos, si la matriz tiene un tamaño considerable, se puede debería que los elementos de entre cada ciclo no estén disponibles en el buffer del microprocesador y los tenga que pedir en la memoria caché reiteradas veces. Lo que ocasiona pérdidas de tiempo por las demoras en pedir elementos que no se encuentran en la memoria, por lo que tampoco se aprovechan los accesos a memoria de manera eficiente.

Para ilustrar esta diferencia, ejecute 2 programas, aprovechando que C es row-major y otro pensando como si fuera column-major.

Matriz de 5000x5000

column-major:

```
$ time ./mal 5000 5000

real    0m1.582s
user    0m1.438s
sys 0m0.144s
```

row-major:

```
$ time ./bien 5000 5000

real    0m1.130s
user    0m0.994s
sys 0m0.136s
```

Matriz 8000x8000

column-major:

```
$ time ./mal 8000 8000

real    0m4.230s
user    0m3.895s
sys 0m0.336s
```

row-major:

```
$ time ./bien 8000 8000

real    0m3.050s
user    0m2.725s
sys 0m0.321s
```

column-major:

```
$ time ./mal 15000 15000

real    0m21.694s
user    0m20.575s
sys 0m1.116s
```

row-major:

```
$ time ./bien 15000 15000

real    0m16.884s
user    0m15.880s
sys 0m1.004s
```

Como vemos en todos los casos los tiempos de ejecución son mayores cuando no aprovechamos las características del lenguaje. Para el caso de matrices muy grandes esto puede ser un gran problema, pero de fácil solución en estos casos. Debemos conocer cómo funciona por debajo el lenguaje que utilizamos.