UNIVERSITAT DE BARCELONA

Sistemas Operativos II

PRÁCTICA 4

4 de noviembre 2020

Autores y NIUB:
María Isabel González Sánchez 20221913
Oriol Saguillo González 20150502

Grupo: 18, B00

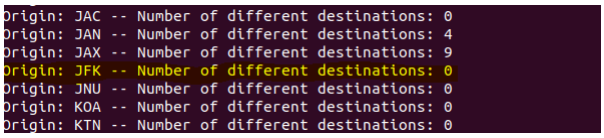
1. Análisis del código

1. Al principio, como funciona la asignación de las 3 letras por aeropuerto, no sabíamos que existía. Una vez realizada una pequeña búsqueda en internet, hemos encontrado cuál es el IATA del aeropuerto de New York: IATA "JFK". Luego hemos compilado y ejecutado el código usando:

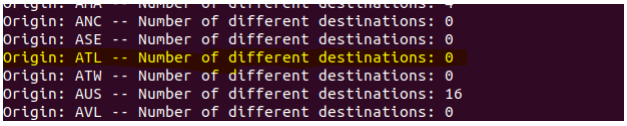
oslab: /Escritorio/SO2/P4/codi$ gcc analisi.c -o analisi

oslab: /Escritorio/SO2/P4/codi$ ./analisi aeroports.csv fitxer_petit.csv

Este sería el resultado que hemos encontrado para el aeropuerto internacional John F. Kennedy (Nueva Jork):



El aeropuerto de Atlanta tiene un IATA más parecido a su nombre real: "ATL".



2. Los 2 aeropuertos con más destinos son:

- Aeropuerto de Chicago Midway con 47 destinaciones:

  Origin: MDW – Number of different destinations: 47

- Aeropuerto de Los Angeles con 54 destinos:

  Origin: LAS – Number of different destinations: 54

Para encontrarlos, hemos derivado la salida por pantalla del código a un archivo txt. Una vez creado, hemos ordenado la última columna del fichero de menor a mayor número de destinos, esto mediante el comando "sort" y teniendo en cuenta que hay destinos de un dígito (de 0 a 9 destinos)y de más de una dígito (de 10 en adelante). Por último, solo nos queda mirar las últimas líneas del archivo para encontrar los vuelos que buscamos. El código será el siguiente:

oslab: /Escritorio/SO2/P4/codi$ ./analisi aeroports.csv fitxer_petit.csv > output.txt

oslab: /Escritorio/SO2/P4/codi$ sort -k 7,7n -k 8,8n output.txt

Una pequeña muestra del resultado puede ser la siguiente, donde mostramos que ordena bien tanto los destinos con un dígito como con varios:

...

Origin: SDF – Number of different destinations: 8

Origin: CMH – Number of different destinations: 9

Origin: JAX – Number of different destinations: 9

Origin: PVD – Number of different destinations: 9

Origin: ELP – Number of different destinations: 10

Origin: MSY – Number of different destinations: 11

Origin: RNO – Number of different destinations: 11

Origin: SJC – Number of different destinations: 11

Origin: SMF – Number of different destinations: 11

Origin: FLL – Number of different destinations: 12

...

3. Vamos a comparar los tiempos de los 2 archivos csv:

- **2007.CSV**

  Origin: YUM – Number of different destinations: 6

  Tiempo para procesar el fichero: 31.518871 segundos

- **2008.CSV**

  Origin: YUM – Number of different destinations: 6

  Tiempo para procesar el fichero: 30.759707 segundos

Como podemos observar, son tiempos altos en ambos ficheros comparados con el del "fitxer_petit.csv"(0.068432 segundos), siendo el de 2008 ligeramente más rápido que el primero. Esto se puede entender viendo los millones de vuelos que guardan (siendo el número de vuelos la resta del número resultante menos 1):

- **VUELOS TOTALES 2007.csv**

  - Número de vuelos totales:

    oslab: /Escritorio/SO2/P4/codi$ cat 2007.csv — wc -l

    7453216

  - Número de destinos totales:

    oslab: /Escritorio/SO2/P4/codi$ awk '{s+=$8}END{print s}' output2007.txt

    5304

- **VUELOS TOTALES 2008.csv**

  - Número de vuelos totales:

    oslab: /Escritorio/SO2/P4/codi$ cat 2008.csv — wc -l

    7009729

  - Número de destinos totales:

    oslab: /Escritorio/SO2/P4/codi$ awk '{s+=$8}END{print s}' output2008.txt

    5362

- **VUELOS TOTALES fitxer_petit.csv**

  - Número de vuelos totales:

    oslab: /Escritorio/SO2/P4/codi$ cat fitxer_petit.csv — wc -l

    10001

  - Número de destinos totales:

    oslab: /Escritorio/SO2/P4/codi$ awk '{s+=$8}END{print s}' output.txt

    822

  - Tiempo para procesar el fichero:

    Tiempo para procesar el fichero: 0.068432 segundos

Ambos ficheros almacenan alrededor de 7 millones de vuelos, a los que además hemos añadido el total de vuelos con destinos para compararlos con el "fitxer_petit.csv".

4. Después de usar el *valgrind* y redirigir la salida a un fichero txt, estos son los errores del código:

- Error 1

  ==3912== Conditional jump or move depends on uninitialised value(s)

  ==3912== at 0x10953A: print_num_flights_summary (in /home/oslab/Escritorio/codi/analisi)

  ==3912== by 0x1099C5: main (in /home/oslab/Escritorio/codi/analisi)

- Error 2

  ==3912== HEAP SUMMARY:

  ==3912== in use at exit: 4,848 bytes in 2 blocks

  ...

  ==3912== LEAK SUMMARY:

  ==3912== definitely lost: 4,848 bytes in 2 blocks

  ...

  ==3912== ERROR SUMMARY: 91809 errors from 1 contexts (suppressed: 0 from 0)

Valgrind nos notifica que ha habido 2 causantes de errores destacables: el primero, valores no inicializados e intento de usarlos, y el segundo, fallo en a memoria dinámica.

En cuanto al primero, nuestro código crea una matriz dinámica con la función *malloc*. A diferencia de su coetánea *calloc*, los valores de dicha matriz no se inicializan a cero. Por lo tanto, al intentar usarlos sin inicializar, hace un salto incondicional porque no conoce esos valores. Para solucionarlo, habría que cambiar las funciones *malloc* por *calloc* para que inicialice todos los valores:

```
ptr = (void **) calloc (nrow, sizeof(void*));
for(i = 0; i < nrow; i++)
    ptr[i] = (void *) calloc (ncol, size);
```

Por otro lado, el segundo error hace referencia a una pérdida definitiva de dos bloques de memoria dinámica de 4848 bytes. Esto se debe a que no se ha liberado dicha memoria, reservada previamente, mediante un *free* de la matriz al acabar el programa. Es decir, no hemos regresado la memoria dinámica que reservamos usando la función *void **malloc_matrix(int nrow, int ncol, size_t size)*. Para arreglarlo, después del bucle for, colocamos la línea "free(matrix);" y soluciona este problema:

```
for(i = 0; i < nrow; i++)
    free(matrix[i]);
free(matrix);
```

Tras estos cambios, vemos que el *valgrind* no se "queja" más:

==6731== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)