UNIVERSITAT DE BARCELONA

*Grado en Ingeniería Informática*

*Facultad de Matemáticas e Informática*

# *PRÁCTICA 4.*

# *PROGRAMACIÓN PARALELA*

*Alumnos: Zixin Zhang - William Sotto*

*Asignatura: Sistemas Operativos II*

*Grupo B*

# Introducción

La programación concurrente y paralela permite la ejecución simultánea de tareas en sistemas multiprocesador y multihilo. A diferencia de la programación secuencial, mejora el rendimiento al distribuir la carga de trabajo, acelerando operaciones intensivas y optimizando el uso de recursos. Esta práctica se centra en convertir el código existente de análisis de vuelos en un programa concurrente mediante la creación y bloqueo de hilos, aprovechando los beneficios de la programación paralela.

# Enunciados

**P1. ¿Qué información muestra? Indica la línea de código que muestre el top 15 de los aeropuertos con más destinaciones de ese año ordenados de mayor a menor.**

La salida obtenida muestra información sobre la cantidad de destinos diferentes para cada aeropuerto en el año 2000. La información se presenta en formato de origen (aeropuerto) seguido del número de destinos diferentes para ese aeropuerto.

***./analisis airports.csv 2000.csv | sort -k8,8nr | head -n 15***

```
$ ./analisis airports.csv 2000.csv | sort -k8,8nr | head -n 15
Origin: ORD -- Number of different destinations: 98
Origin: ATL -- Number of different destinations: 97
Origin: MSP -- Number of different destinations: 92
Origin: DFW -- Number of different destinations: 90
Origin: DTW -- Number of different destinations: 83
Origin: STL -- Number of different destinations: 80
Origin: IAH -- Number of different destinations: 79
Origin: CLT -- Number of different destinations: 70
Origin: PIT -- Number of different destinations: 70
Origin: EWR -- Number of different destinations: 67
Origin: PHX -- Number of different destinations: 67
Origin: DEN -- Number of different destinations: 65
Origin: LAS -- Number of different destinations: 65
Origin: CVG -- Number of different destinations: 64
```

**P2. ¿Cuánto tiempo tarda en ejecutarse cada uno? ¿Con qué comando podrías averiguarlo (ignorando el tiempo que ya devuelve el programa)? Para estos mismos dos ficheros (1990 y 2007)**

Para medir el tiempo de ejecución de un programa en Linux, usamos el comando **time** antes de cada ejercicio. "Real" indica el tiempo total transcurrido, "user" indica el tiempo de CPU en el espacio de usuario, y "sys" indica el tiempo de CPU en el espacio del kernel.

***time ./analisis airports.csv 1990.csv***

```
real    0m5,232s
user    0m5,074s
sys     0m0,124s
```

***time ./analisis airports.csv 2007.csv***

```
real    0m7,765s
user    0m7,488s
sys     0m0,220s
```

**P3. ¿Qué comandos puedes utilizar para calcular el número de vuelos que contiene cada uno de ellos?¿Cuántos vuelos contienen cada uno de ellos?**

Para saber cuántos vuelos hay podemos contar la cantidad de líneas del archivo .csv en cuestión y luego restarle 1, ya que estos archivos tienen una línea que corresponde a la información de cada columna.

Para 1990.csv: echo $(($(wc -l < 1990.csv) - 1)) => 5270893 | Para 2007.csv echo $(($(wc -l < 2007.csv) - 1)) => 7453215

**P4. ¿Cuáles son las funciones que más tardan en ejecutarse? ¿Cuál es el porcentaje de tiempo que ¿ocupa la ejecución de cada una de las funciones?**

**time valgrind --tool=callgrind ./analisis airports.csv file_test.csv**

```
Tiempo para procesar el fichero: 1.720872 segundos
==2006==
==2006== Events    : Ir
==2006== Collected : 168727172
==2006==
==2006== I   refs:      168,727,172

real    0m2.025s
user    0m2.015s
sys     0m0.010s
```

Luego de crearnos el callgrind.out podemos ver el peso de cada llamada de función que se ha necesitado para file_test.csv. La columna incl. indica el porcentaje de tiempo que se ha necesitado para cada funcion.

| Incl. | Self | Called | Function | Location |
|---|---|---|---|---|
| 100.00 | 0.00 | (0) | 0x0000000000020290 | ld-linux-x86-64.so.2 |
| 99.91 | 0.00 | 1 | (below main) | analisis |
| 99.90 | 0.00 | 1 | main | analisis |
| 98.49 | 0.29 | 1 | read_airports_data | analisis |
| 84.67 | 32.32 | 20 000 | get_index_airport | analisis |
| 52.64 | 4.32 | 3 363 147 | 0x0000000000109180 | (unknown) |
| 12.20 | 11.52 | 10 000 | extract_fields_airport | analisis |

**P5. ¿Qué errores se observaron a través de la salida del Valgrind? ¿Cómo los has corregido?**

Primero, podemos cargar primero de nuevo análisis, para tener la opción de depuración:

**gcc -g -o analisis analisis.c**

Para poder ver las fugas de memoria y otros problemas debemos introducir la siguiente línea

**valgrind -s --leak-check=full ./analisis airports.csv 2000.csv**

Después de ver la salida podemos ver que se han perdido bytes:

```
LEAK SUMMARY:
   definitely lost: 4,848 bytes in 2 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

Nos señala que ya no podemos acceder a estos dos bloques de memoria y ya no podemos recuperarlas (still reachable=0)

Para solucionar este problema podemos ir a las líneas que nos señala el heap Summary y liberar la memoria en cuestión antes de que el programa finalice.

```
HEAP SUMMARY:
    in use at exit: 4,848 bytes in 2 blocks
  total heap usage: 613 allocs, 611 frees, 383,456 bytes allocated

2,424 bytes in 1 blocks are definitely lost in loss record 1 of 2
   at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x1092D0: malloc_matrix (analisis.c:26)
   by 0x109977: main (analisis.c:316)

2,424 bytes in 1 blocks are definitely lost in loss record 2 of 2
   at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x1092D0: malloc_matrix (analisis.c:26)
   by 0x10998F: main (analisis.c:317)
```

La salida de Valgrind indica que hay dos bloques de memoria (totalizando 4,848 bytes) que fueron asignados con **malloc** y no fueron liberados antes de que el programa finalizará. Estos bloques de memoria se asignan en la función **malloc_matrix** en la línea 26 de **analisis.c** y se pierden definitivamente en la función **main** en las líneas 316 y 317.

```
void free_matrix(void **matrix, int nrow)
{
  int i;

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

  free(matrix); //Línea añadida para evitar
                //perdida de memoria
}
```

El problema es que el la función free matrix, la matrix en sí no se ha liberado como sus índices, añadimos este free para solucionar el problema de memoria.

```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 613 allocs, 613 frees, 383,456 bytes allocated

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

**P6. ¿Qué variables debería pasar por argumento el hilo principal a los secundarios?**

Cada hilo tiene la cantidad de líneas que le toca leer, definidas *ThreadArgs* como *n_lines*, los aeropuertos, num_flights y el file de lectura (fp). Con toda esta información los hilos secundarios pueden calcular los números de vuelos.

**P7. ¿Qué partes del código (secciones críticas) has protegido para evitar interferencias? Justifica tu respuesta.**

Las secciones críticas son aquellas donde se actualiza la matriz *num_flights* ya que múltiples hilos pueden intentar acceder y modificar los mismos elementos de la matriz simultáneamente. En nuestro código, este acceso en la sección crítica se encuentra en la función *process_data_block*. Para evitar interferencias, teníamos que proteger esta sección crítica utilizando mecanismos de exclusión mutua, como los mutex proporcionados por la biblioteca *pthread*.

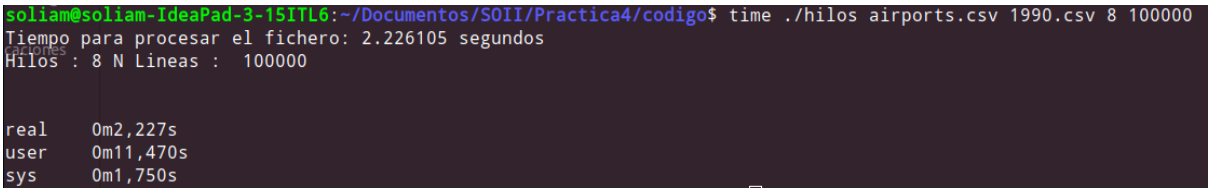
**P8. Muestra una tabla mostrando los tiempos de ejecución para diferente número de hilos (H = 1, 2, 3, 4, 6 y 8), así como diferentes líneas por bloque (N = 1, 10, 100, 1.000, 10.000, 100.000). Comenta los resultados.**

| hilos\líneas | 1 | 10 | 100 | 1000 | 10.000 | 100.000 |
|---|---|---|---|---|---|---|
| **1** | 5,1834 s | 5,1209 s | 5,0816 s | 5,0280 s | 5,0962 s | 5,0702 s |
| **2** | 3,0773 s | 3,0771 s | 3,1001 s | 3,0922 s | 3,054 s | 3,096 s |
| **3** | 2,7813 s | 2,819 s | 2,7893 s | 2,8417 s | 2,795 s | 2,7983 s |
| **4** | 2,5597 s | 2,5649 s | 2,5613 s | 2,5263 s | 2,583 s | 2,5978 s |

| | | | | | | |
|---|---|---|---|---|---|---|
| **6** | 3,2078 s | 2,1866 s | 2,5425 s | 2,3551 s | 2,1661 s | 2,1778 s |
| **8** | 2,3703 s | 2,46 s | 2,2073 s | 2,171 s | 2,4293 s | 2, 24 s |

Hemos ejecutado 3 veces para cada permutación y sacado la media para los valores de la tabla.

Una ejecución normal de nuestro programa se ve de la siguiente manera:




Viendo los valores obtenidos en la tabla hemos llegado a la conclusión de que el número de líneas que tiene cada hilo no llega a afectar realmente el tiempo de ejecución, al ir repitiendo la cantidad de líneas dadas al hilo al acabar, es como si no le hubiésemos asignado este número de líneas. Es decir, cuando un hilo acaba su número de líneas asignado, vuelve a leer este hasta que se acabe el fichero. Es como si solo le dijéramos que vaya leyendo las líneas directamente.

Por ende, si queremos reducir el tiempo de ejecución promedio, lo mejor es aumentar el número de hilos que nos permita el Sistema Operativo.